ALGOL PRIMER

FOR

BURROUGHS B 6700

Peter V. de Souza
&
David C. E. Manley

ALGOL PRIMER

FOR

BURROUGHS B6700

Peter V. de Souza
&
David C.E. Manley

(both of Department of Preventive and Social Medicine - Dunedin, University of Otago)

First published 1973
Reprinted 1974
Reprinted 1974

Published by Prevsoc Publications, Dunedin.

Printed in New Zealand by John McIndoe Ltd., Dunedin.

PREFACE

There does not appear to be a suitable elementary textbook covering the basic concepts of Burroughs' implementation of the ALGOL programming language on their B6700 computer. We have attempted to fill the gap with this book which is designed to help the student learning ALGOL for the first time, but it is hoped that it will also be of use to those who are transferring from other implementations of the language.

At the time of writing, version 2.5 of the B6700 system software had just been released; since each new release of the software usually contains minor modifications to the ALGOL compiler, some aspects of the features described in this book may change with subsequent releases. For this reason a sheet has been provided at the end of the book on which readers may make general comments or suggest improvements and amendments for the future.

While this book was being written one of the authors, P.V. de Souza, was supported by a grant from the Medical Research Council of New Zealand and their help is gratefully acknowledged.

University of Otago, 1973

Peter V. de Souza
David C.E. Manley

CONTENTS

CHAPTER 1

INTRODUCTION TO ALGOL

1.1 A Simple ALGOL Program

Consider a program which tells the computer to read in 2 integers at the card reader and write out their mean at the line printer. One such program is the following:

```
BEGIN
   FILE CARD(KIND=READER);
   FILE LP(KIND=PRINTER);
   INTEGER A, B;
   REAL MEAN;
   READ(CARD, /, A, B);
   MEAN := (A+B)/2;
   WRITE(LP, /, MEAN)
END.
```

Program 1.

This program reads in the values of 2 integers called A and B, finds their mean using the formula $\frac{A+B}{2}$ and writes out the value of the mean.

1.2 Identifiers

In the above program A, B and MEAN are the names of 3 variables. The technical word for a name is an 'identifier' since it identifies which variable is being manipulated.

In general any uninterrupted sequence of letters and digits which begins with a letter is a valid identifier. Certain words however, have a special meaning in ALGOL and may not be used as identifiers. These 'reserved' words are:

ALPHA	FALSE	PROCEDURE
ARRAY	FILE	REAL
BEGIN	FOR	STEP
BOOLEAN	FORMAT	SWITCH
COMMENT	GO	TASK
CONTINUE	IF	THEN
DIRECT	INTEGER	TRUE
DO	LABEL	UNTIL
DOUBLE	LIST	VALUE
ELSE	LONG	WHILE
END	OWN	ZIP
EVENT	POINTER	

Furthermore certain identifiers can be used but have restrictions associated with them and are best avoided. These are:

ACCEPT	EXCHANGE	MONITOR	RESIZE
AND	EXTERNAL	MUX	REWIND
ATTACH	FILL	NEQ	RUN
BY	FORWARD	NO	SCAN
CALL	GEQ	NOT	SEEK
CASE	GTR	ON	SET
CAUSE	IMP	OPEN	SKIP
CLOSE	IN	OR	SORT
DEALLOCATE	INTERRUPT	OUT	SPACE
DEFINE	IS	PICTURE	SWAP
DETACH	LB	PROCESS	THRU
DISABLE	LEQ	PROCURE	TIMES
DISPLAY	LIBERATE	PROGRAMDUMP	TO
DIV	LINE	RB	WAIT
DUMP	LOCK	READ	WHEN
ENABLE	LSS	RELEASE	WITH
EQL	MERGE	REPLACE	WRITE
EQV	MOD	RESET	

and the names of all the intrinsic functions. (More is said about the intrinsics in section 1.5).

Example

Valid identifiers	Invalid identifiers
Y	5N
NUMBER	NO.
N5	F(X)
P64QL8	BIG RAT
APLUSB	BEGIN
TEATIME	TEA-TIME
FX	INTEGER

Thus program 1 would have been just as correct using the identifiers X,Y, Z instead of A,B, MEAN. It is good programming practice however to choose identifiers, where possible, which convey information about the corresponding variables. In program 1, MEAN is a better identifier than Z because MEAN explains what purpose the variable is being used for whereas Z does not.

1.3 Declarations

In general when a user's program is submitted to a computer for running, a special program called a compiler converts the user's program into something which the computer can understand more easily. Now since the compiler needs to know all about the variables used in the user's program, they must be introduced to the compiler before they are used for the first time. This introduction is carried out by a 'declaration'.

In the 4th line of program 1 above, the variables A, B are declared to be of type 'integer' i.e. A and B will only ever hold integral values throughout the program. These integral values may be positive, negative or zero. If a variable may take non-integral values (e.g. 2.3, - 5.5 etc.) then it is declared to be of type 'real' and the numbers it holds are said

to be real numbers as opposed to integers. In the 5th line of program 1, MEAN is declared to be of type real since it will not necessarily hold integral values.

Notice that several variables can be declared together simply by separating them with commas. A semicolon is used at the end of the list of variables being declared to separate that declaration from a following declaration or a following statement. For examples of this, see lines 4 and 5 of program 1. Further examples are:

```
INTEGER ONE, TWO, AG, MAN;
REAL THEE, ME, M99;
```

There are 2 other declarations in program 1. Lines 2 and 3 are FILE declarations but these are dealt with later in section 2.2.

1.4 Arithmetic Operators

The following arithmetic operators are available in B6700 ALGOL:

**	=	exponentiation
*	=	multiplication
/	=	division
+	=	addition
-	=	subtraction
DIV	=	integer division
MOD	=	modulo division

1. Exponentiation

Y ** Z is equivalent to the usual mathematical notation Y^Z. Note, however, the following restrictions:

(a) Y ** Z is undefined if Y = 0 and Z $\leq$ 0 since 0^Z is undefined for Z $\leq$ 0.

(b) If Y < 0 and Z is of type real then Y ** Z is undefined if Z $\neq$ 0 since Y^Z is usually complex in this case.

2. Multiplication

A * B means A multiplied by B. Note that A * B is the only acceptable form of multiplication in ALGOL i.e. A.B and AB are not allowed. Thus the ALGOL equivalent of 2Y is 2 * Y and the ALGOL equivalent of 2(A+B) is 2*(A+B).

3. Division

A/B means A divided by B.

A DIV B returns the integer obtained by truncating A/B at the decimal point. Thus since 3/2 = 1.5, 3 DIV 2 = 1. Similarly 3 DIV -2 = -3 DIV 2 = -1.

A MOD B = A-(B*(A DIV B)). Thus A MOD B returns the remainder obtained after dividing A by B. For example 3 MOD 2 = 1, 3 MOD -2 = 1, -3 MOD 2 = -1.

Note that spaces should be used to separate DIV and MOD from both operands. Note also that A/B, A DIV B, and A MOD B are all undefined if B = 0.

4. Addition and Subtraction

A + B is simply A plus B and
A - B is A minus B.

5. Precedence of Arithmetic Operators

In an expression containing more than 1 arithmetic operator the operations are performed in a sequence which is determined by the precedence of the operators involved. The order of precedence is:

(1) **
(2) * / MOD DIV
(3) + -

Thus any operations involving the exponentiation operator ** are done first etc. When operators have the same order of precedence the operations are performed going from left to right. Round brackets can be used in normal mathematical fashion to override the usual order of precedence.

Examples

ALGOL expression	Mathematical equivalent
A+B/2	$A+\frac{B}{2}$
(A+B)/2	$\frac{A+B}{2}$
X**2-Y/3*Z	$X^2-\frac{Y.Z}{3}$
(X**2-Y)/3*Z	$\frac{(X^2-Y)Z}{3}$
(X**2-Y)/(3*Z)	$\frac{(X^2-Y)}{3Z}$
A*(-B)	A.-B

Note that 2 operators must never be adjacent.

6. Types of Results of Arithmetic Operations

operand on left	operand on right	value resulting from the operator + - *	/	DIV	MOD	**
Integer	Integer	Note 1	Real	Integer	Integer	Note 2
Integer	Real	Real	Real	Integer	Real	Note 3
Real	Integer	Real	Real	Integer	Real	Note 3
Real	Real	Real	Real	Integer	Real	Note 3

Note 1: the resulting value is of type integer unless the absolute value of the result is 2^{39} or greater in which case it is of type real.

Note 2: The resulting value is of type integer unless the absolute value of the result is 2^{39} or greater or the operand on the right is negative in which cases the result is of type real.

Note 3: The resulting value is of type real unless the operand on the right is zero in which case it is of type integer.

1.5 Arithmetic Intrinsics

The following arithmetic intrinsic functions are available in B6700 ALGOL. 'AE' is used to indicate an arithmetic expression.

Function	Type of Result	Meaning
ABS(AE)	REAL	Absolute value of AE.
SIGN(AE)	INTEGER	+1 if AE > 0; 0 if AE = 0, -1 if AE < 0.
SIN(AE)	REAL	Sine of AE radians.
COS(AE)	REAL	Cosine of AE radians.
TAN(AE)	REAL	Tangent of AE radians.
LOG(AE)	REAL	Log. to the base 10 of AE. (AE > 0)
LN(AE)	REAL	Natural log. of AE i.e. $\text{Log}_e(AE)$. (AE > 0).
EXP(AE)	REAL	e raised to the power of AE i.e. e^{AE}.
SQRT(AE)	REAL	Positive square root of AE. ($AE \geq 0$).
ENTIER(AE)	INTEGER	The largest integer $\leq$ AE.
INTEGER(AE)	INTEGER	The value of AE rounded to the nearest integer.

Notes 1. The argument of SQRT must be non-negative.

2. The arguments of LOG and LN must be greater than 0.

3. The arguments of all the arithmetic intrinsics must be enclosed in round brackets.

4. ENTIER does not perform simple truncation.
For example, ENTIER (2.6) = 2
ENTIER (-2.6) = -3
ENTIER (-0.01) = -1

5. There are many other intrinsic functions available and these can be found in BURROUGHS B6700/B7700 EXTENDED ALGOL LANGUAGE INFORMATION MANUAL, Pages 6-11 to 6-20.

1.6 Statements

Lines 1 and 9 of program 1 (section 1.1) i.e. 'BEGIN' and 'END' do little more than signify the beginning and end of the program - they can be thought of as punctuation symbols. Lines 2 - 5 we have seen are declarations and do nothing more than convey information to the compiler about the variables used. It is important to realise that declarations are non-executable for this distinguishes them from statements which actually tell the computer to do something. The other 3 lines i.e. 6 - 8, are statements and are executed by the computer. Lines 6 and 8 are read and write statements respectively and line 7 is an assignment statement. All these statements are discussed in Chapter 2.

1.7 Structure of Simple ALGOL Programs

In looking back at program 1 we see that it consists of a BEGIN, 4 declarations, 3 statements and an END. Notice that semicolons are used to separate each declaration from the declaration or statement which follows it and each statement from the following statement. (In the examples throughout this book a semicolon is used after the last ALGOL statement. This semicolon is not part of the last statement but merely indicates that in a program a semicolon would usually be required as there would normally be subsequent statements.) No semicolon is necessary between the last statement and the END since END is not a statement. A full stop is placed after the END in the last line of the program.

The structure of simple ALGOL programs can be summarised as follows:

```
BEGIN
   D;  )
   .   |
   .   } one or more declarations
   .   |
   D;  )
   S;  )
   .   |
   .   } one or more statements
   .   |
   S   )
END.
```

CHAPTER 2

STATEMENTS

2.1 Assignment Statements

The general form of an assignment statement is

```
identifier := expression;
```

in which the variable named on the left is assigned the value of the expression on the right.

Examples

1. `MEAN := (A+B)/2 ;`

2. `A := A+1 ;`

The value of a variable is only altered when it appears on the left hand side of an assignment. The values of variables appearing on the right hand side are used but not altered by the assignment statement. When a variable appears on both sides of an assignment, the expression on the right is evaluated using the present value of the variable (i.e. before the assignment) and the result is then assigned to the variable as its new value.

Thus if A = 3 then `A := A + 1` is equivalent to

```
A := 3 + 1
```

i.e. `A := 4`

Note that `A - 2 := 6` is not allowed since `A - 2` is not an identifier but `A := 6 + 2` is allowed and amounts to the same thing.

2.2 Simple Input/Output Statements

As this is an introduction to ALGOL the only input considered will be that from the card reader and the only output will be at the line printer, i.e. it is assumed that when variable values are read in they are read in from cards at the card reader and when results are output they are written onto paper at the line printer. In B6700 ALGOL however, information is always read and written onto 'files'. In order to read or write information then, it is necessary to declare the required files to tell the compiler all about the types of files that will be used throughout the program.

The general form of a declaration of a card reader file is:

```
FILE filename (KIND = READER);
```

where 'filename' is the identifier of the file used throughout the program and can be chosen by the programmer using the usual rules for identifiers.

Similarly the declaration of a line printer file has the general form:

```
FILE filename (KIND = PRINTER);
```

Line 2 of program 1 (section 1.1) declares a card reader file called CARD and line 3 declares a line printer file called LP.

The general form of read/write statements in B6700 ALGOL is:

```
READ
 or      (filename, format, list of variables for reading or writing);
WRITE
```

where 'filename' is the identifier of the appropriate file as given in its declaration; 'format' can take many forms but initially it will always be '/' which tells the computer that the data is to be read/written in 'free-field' form; the list of variables is just the list of identifiers, separated by commas, which either contain the information to be written or are to contain the information to be read in. When data is being read from cards in free-field form the numbers on the cards should be separated by commas; when data is printed in free-field form each number is followed by a comma.

Looking back at program 1 again, we see that line 6 is a READ statement which tells the computer to read in the values of A and B in free-field form and to use the file called CARD which has been set up as a card reader file. Line 8 is a WRITE statement and tells the computer to write out the value of MEAN in free-field form and to use the file called LP which is a line printer file.

Finally note that each READ statement causes a new card to be read i.e. reading does not continue from where the last READ statement left off, and each WRITE statement causes a new line to be taken at the line printer.

2.3 Boolean Expressions

Just as the result of an arithmetic expression is an arithmetic value, so the result of a Boolean expression is a Boolean value of which there are only two: TRUE and FALSE.

A simple Boolean expression has the general form

AE relational operator AE

where 'AE' denotes an arithmetic expression and the 'relational operators' are <, >, =, < =, > =, ¬ = meaning less than, greater than, equal to, less than or equal to, greater than or equal to, and not equal to. Alternative forms of these operators are LSS, GTR, EQL, LEQ, GEQ, NEQ.

Example If I is an integer with value 5 then

Boolean Expression	Value
I > 5	FALSE
I < 5	FALSE
I = 5	TRUE
I LEQ 5	TRUE
I GEQ 5	TRUE
I NEQ 5	FALSE
2*I = 5+3	FALSE
I+5 <= 25-I	TRUE

The reserved words TRUE and FALSE are called Boolean primaries and always have values TRUE and FALSE respectively - their values cannot be changed.

The logical operators NOT, AND, OR can be used to change or combine the values of Boolean expressions. (The symbol ¬ may be used instead of the word NOT.)

NOT must be followed by a Boolean expression (or primary) and negates in the logical sense the value of that expression.

Examples The Boolean value of NOT TRUE is FALSE.
The Boolean value of NOT (I < 5) is TRUE (assuming still I = 5)

AND and OR are used to combine the values of two Boolean expressions and return a single Boolean value. The general form for their use is

Boolean expression $\begin{matrix}\text{AND}\\\text{OR}\end{matrix}$ Boolean expression.

When using AND, the result is TRUE if both expressions are true otherwise it is FALSE. When using OR, the result is TRUE if one or both expressions are true otherwise it is FALSE. Thus if I=5,

```
I > 5 AND I = 5 is FALSE
I < 15 AND I > 2 is TRUE
I > 5 OR I = 5 is TRUE
I = 6 OR I = 7 is FALSE
2*I < 25 OR I + 1 > 2 is TRUE
```

When Boolean expressions are evaluated, the order of precedence is as follows:

1. Arithmetic Expressions
2. Relations
3. NOT
4. AND
5. OR

When operators have the same order of precedence they are evaluated from left to right. As with arithmetic expressions, round brackets may be used to override the order of precedence.

Example

If A = 5, B = 3, C = 1 and D = 4 then

(A * B > C + D OR D < C - A) AND A > B is evaluated as follows:

```
(15 > 5 OR 4 < -4) AND 5 > 3
(TRUE   OR FALSE ) AND TRUE
       TRUE        AND TRUE
                   TRUE
```

Besides integer and real, a variable may be of type Boolean but may then hold only the values TRUE and FALSE. The general form of a Boolean declaration is

BOOLEAN variable list;

Example BOOLEAN OK, FLAG;

A Boolean variable may be assigned Boolean values in the same way that an integer variable can be assigned integer values.

Example After the declaration above, permissable statements would be:

```
OK := TRUE;
FLAG := I > 5 AND I < 16;
```

Boolean variables can be very useful in connection with conditional statements as will be seen later.

2.4 IF Statements

IF statements have 2 general forms:

1. IF Boolean expression THEN statement;
2. IF Boolean expression THEN statement 1 ELSE statement 2;

IF statements are an example of 'conditional statements' since their outcome is conditional upon the value of a Boolean expression.

Consider the first type of IF statement. If the 'Boolean expression' is true then the 'statement' is executed and if it is false then the 'statement' is not executed.

Example Before dividing A by B in a program it may be desirable to check that B is non-zero and not do the division if it is. One way of doing this might be

```
IF B NEQ 0 THEN C := A/B;
```

If B = 0 then the division would not be performed and C would not be assigned a value and program control would pass to the next statement in the program.

In the second form of IF statement, if the 'Boolean expression' is true then 'statement 1' is executed and if it is false then 'statement 2' is executed. Note that there is no semicolon between the first statement and the ELSE.

Example If B = 0 THEN C := A ELSE C := A/B;

This time if B = 0 then C is assigned the value of A and if B ≠ 0 then C is assigned the value of A/B. In either case only one assignment is made to C and after it program control passes to the next statement after the IF statement.

In an IF statement the statements to be executed may themselves be IF statements i.e. IF statements may be 'nested'. Hence it is possible to have an IF statement of the form

```
IF Boolean expression THEN IF Boolean expression THEN statement;
```

If the first Boolean expression is true then the secondary IF statement is executed. If the first Boolean expression is false then the secondary IF statement is completely ignored and control passes to the next instruction. If the secondary IF statement is executed its Boolean expression is evaluated; if that Boolean expression is true then the final statement is executed but if it is false the final statement is ignored and control passes to the next

instruction. Hence both Boolean expressions must be true if the statement at the end is to be executed.

Example IF A < 10 THEN IF A > 5 THEN B := 1;

The assignment to B is only performed if A < 10 is true and A > 5 is true. Notice that an alternative way of obtaining the same end is

```
IF A < 10 AND A > 5 THEN B := 1;
```

The example above consists of an IF statement of the first kind within another IF statement of the first kind. (Recall that there are 2 forms of the IF statement).

An ambiguity arises when either of the IF statements is of the second kind.

Example 1

```
IF A < 10 THEN IF A > 5 THEN B := 1 ELSE B := 2;
```

Without some guiding rule it is not clear which THEN the ELSE corresponds to. To avoid this ambiguity there is a simple rule which states that the ELSE is matched with the last THEN which does not already have an ELSE. Thus in this example the ELSE goes with the second THEN and therefore B is assigned a value only if A < 10. The full breakdown of the statement is as follows

1) If A $\geq$ 10 no assignment is made,

2) If A < 10 and A > 5 B is assigned the value 1,

3) If A < 10 and A $\leq$ 5 B is assigned the value 2.

Example 2

```
IF A < 10 THEN B := 1 ELSE IF A < 20 THEN B := 2 ELSE B := 3;
```

This assigns B the value of

1 if A < 10,

2 if 10 $\leq$ A < 20,

3 if A $\geq$ 20.

Example 3

```
IF X < 20 THEN IF A < 10 THEN B := 1 ELSE B :=  2 ELSE B := 3;
```

This time B is assigned the value of

1 if X < 20 and A < 10,

2 if X < 20 and A $\geq$ 10,

3 if X $\geq$ 20.

Note that the following construction is not allowed:

```
IF A < B < C THEN statement;
```

A < B < C is really 2 conditions viz. A < B and B < C. Correct versions are

```
IF A < B AND B < C THEN statement;
```

or

```
IF A < B THEN IF B < C THEN statement;
```

Boolean variables can be used as well as explicit Boolean expressions in IF statements.

Example If OK is a Boolean variable then

```
OK := I > 1  AND  I < 6;
IF OK THEN A := A+1;
```

is equivalent to

```
IF I > 1  AND  I < 6  THEN  A := A+1;
```

It may be required to execute more than one statement if a condition is true. For example if B is non-zero it may be wished to assign a value, X, to A, calculate A/B and write out the answer. With the techniques already described this can only be done using 3 IF statements one after the other:

```
IF  B NEQ 0 THEN A := X;
IF  B NEQ 0 THEN C := A/B;
IF  B NEQ 0 THEN WRITE (LP, /, C);
```

Obviously this is hopelessly laborious particularly if many statements are dependent on the condition and so a method is required which enables the programmer to group several statements together so that a whole group can be controlled by one IF statement. Fortunately ALGOL permits the programmer to group statements together into what is called a 'compound statement'. In contrast single statements on their own such as have been met already are called 'simple statements'.

2.5 Compound Statements

The general form of a compound statement is

```
BEGIN
   S;
   S;
   .
   .      } 2 or more statements
   .
   S
END;
```

i.e. a compound statement consists of 2 or more statements enclosed by the symbols BEGIN and END. The symbols BEGIN and END have no significance other than to indicate to the compiler where the compound statement begins and ends. The statements within the compound statement are separated from each other by semicolons; since BEGIN and END are not statements no semicolon is required between the BEGIN and the first statement or between the last statement and the END. A full stop is not used after the END of a compound statement - a full stop is only used after the last END of a program.

Example

```
BEGIN
   A := X;
   C := A/B;
   WRITE (LP, /, C)
END;
```

Now although a compound statement actually consists of several statements it is considered by the compiler to be just one statement and is treated just like a simple statement. Hence a compound statement can appear anywhere in an ALGOL program that a simple statement can. For example, the first form of the IF statement was given in section 2.4 as

IF Boolean expression THEN statement;

The 'statement' may be either simple or compound here.

Example

```
IF B NEQ 0 THEN
BEGIN
   A := X;
   C := A/B;
   WRITE (LP, /, C)
END;
```

Notice that this is still nothing more than an IF statement but now the statement controlled by the Boolean expression is a compound statement. If B ≠ 0 then the compound statement is executed i.e. all 3 simple statements within it are executed. If B = 0 then the compound statement is ignored i.e. all 3 simple statements are ignored.

The statements within a compound statement may themselves be compound.

Example

```
IF B NEQ 0 THEN
BEGIN
   A := X;
   C := A/B;
   IF B = 1 THEN
   BEGIN
      WRITE (LP, /, X);
      B := 10
   END;
   WRITE (LP, /, C)
END;
```

Here the compound statement controlled by the Boolean expression B NEQ 0 consists of 3 simple statements and 1 compound statement. If B = 0 the whole of the compound statement is ignored i.e. everything between the outer BEGIN and END is ignored. If B ≠ 0 the 3 simple statements within the outer compound statement will be executed but the inner compound statement will only be executed if B = 1.

It was stated in the last section that in nested IF statements every ELSE is matched with the last THEN which does not already have an ELSE. Nested IF statements should not be confused with the situation where a compound statement containing an IF statement is controlled by an IF statement. A THEN within a compound statement cannot have a corresponding ELSE outside that compound statement.

Example

```
IF B NEQ 0 THEN
BEGIN
   C := B;
   IF X = 1 THEN WRITE (LP, /, B)
END
ELSE WRITE (LP, /, A);
```

This statement is of the form

IF Boolean expression THEN statement 1 ELSE statement 2;

where 'statement 1' is a compound statement and 'statement 2' is simple. Written like this it is clear that the ELSE goes with the THEN before statement 1 even though this is not the last THEN. Thus in this example, if B ≠ 0 C is assigned the value of B and if X = 1 also then the value of B is output; if B ≠ 0 and X ≠ 1 nothing is output; if B = 0 then the value of A is output.

2.6 GO TO Statements and Labels

A statement in an ALGOL program may have a label attached to it by which the statement may be identified. The label is written to the left of the statement and followed by a colon.

Example `START: MEAN:= (A + B)/2;`

Here the statement has been labelled with START.

If labels are used in a program then like all other identifiers they must be declared. The general form of a label declaration is

LABEL list of identifiers;

Hence in the case above, the appropriate declaration would be

LABEL START;

Labels make it possible for program control to jump around within a program so that the statements are not executed consecutively. This jumping is done using GO TO statements of which the general form is

GO TO label;

Example

```
        READ (CARD, /, A, B);
START:  MEAN := (A + B)/2;
        WRITE (LP, /, MEAN);
        READ (CARD, /, A, B);
        IF A NEQ 0 OR B NEQ 0 THEN GO TO START;
```

If lines 6 - 8 of program 1 (section 1.1) are replaced by this piece of code then pairs of A, B are read in and processed until a pair of zeros is read, in which case the program will terminate (assuming that the first pair read in are not both zero). Whenever A, B are read such that at least one of them is non-zero control is passed to the line labelled START which computes the mean and then recycles.

A piece of program being used over and over again like this is called iteration.

2.7 FOR Statements

Another way in which iteration can occur is by using FOR statements. The general form of a FOR statement is:

```
FOR control variable := for-list DO statement;
```

where 'statement' may be any sort of statement including a compound statement, this statement being executed once for each value of the 'control variable' in the 'for-list'. Note that a compound statement must be used when more than one statement is to be executed iteratively.

Example `FOR I := 1, 2, 3 DO A := A + I;`

Here the control variable is I, the for-list is 1, 2, 3 and the statement is A := A + I.

The effect of this is to execute the statement 3 times with I having the values 1, 2, 3 respectively, i.e. it is an alternative way of writing

```
A := A + 1;
A := A + 2;
A := A + 3;
```

Now it may be desirable to execute the statement A := A + I, 20 times instead of 3 so that a very long for-list would be required:

```
FOR  I := 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
DO   A := A + I;
```

Instead of writing out all 20 elements of the for-list, however, an abbreviated version can be used. This abbreviated version has the form:

```
AE STEP AE UNTIL AE
```

where 'AE' represents an arithmetic expression.

Example

```
FOR I := 1 STEP 1 UNTIL 6 DO A := A + I;
```

This is equivalent to

```
A := A + 1;
A := A + 2;
A := A + 3;
A := A + 4;
A := A + 5;
A := A + 6;
```

so that if A started off with the value 0 it would end up by holding the sum of the first 6 integers. In this example the for-list is

```
1 STEP 1 UNTIL 6
```

i.e. the control variable I, is initialised with the value 1 and incremented by a step of 1 until I = 6 and then control passes to the next instruction in the program.

Generally,

```
FOR I := A STEP B UNTIL C DO statement;
```

causes I to be initialised with the value A and incremented by an amount B until the limiting value C is passed. The 'statement' is not executed when I passes the limiting value of C. Note that if the control variable is initialised with a value already past the limiting value C then the statement is never executed.

Example

```
FOR J := 10 STEP 1 UNTIL 8 DO statement;
```

The 'statement' is never executed here.

The control variable in a FOR statement may be an integer or a real variable and must be declared like any other variable. It is advisable to make the control variable an integer as this avoids errors due to rounding and ensures that the statement is executed the correct number of times. Any FOR statement involving a real control variable can usually be rewritten using an integer variable.

Example

Instead of

```
FOR X := 0.0 STEP 0.1 UNTIL 2.0 DO A := A + X;
```

the following statement may be used:

```
FOR I := 0 STEP 1 UNTIL 20 DO A := A + I/10;
```

Step sizes may be positive or negative.

Examples

1. FOR I := 1 STEP 2 UNTIL 10 DO statement;

causes the 'statement' to be executed with values of I = 1, 3, 5, 7, 9 (it is not executed with I = 11).

2. FOR I := 1 STEP 2 UNTIL 11 DO statement;

causes the 'statement' to be executed with values of I = 1,3,5,7,9,11. It is executed with I = 11 because with this value, I has not gone past the limiting value.

3. FOR J := 10 STEP -2 UNTIL 1 DO statement;

causes the 'statement' to be executed with values of I = 10, 8, 6, 4, 2 (it is not executed with J = 0).

The control variable in a FOR statement need not necessarily appear in the statement being operated on by the FOR clause.

Example The following piece of program tells the computer to read in 10 pairs of numbers A, B and output their means.

```
FOR I := 1 STEP 1 UNTIL 10 DO
BEGIN
  READ (CARD, /, A, B);
  MEAN := (A + B)/2;
  WRITE (LP, /, MEAN)
END;
```

Notice that the statement being operated on in this iteration is a compound statement and that no use is made of the control variable within the compound statement. Here the control variable serves only the purpose of counting the number of times the statement is executed. The compound statement is used to group the 3 simple statements together so that all 3 are executed iteratively.

It is possible to have a for-list which comprises a list of values for the control variable and an implied list using STEP - UNTIL.

Example

```
FOR COUNT := 1,7,8,10 STEP 1 UNTIL 15, 19, - 6 DO statement;
```

In this example the for-list is made up of 3 parts

1) 1, 7, 8,
2) 10 STEP 1 UNTIL 15, (i.e. 10, 11, 12, 13, 14, 15,)
3) 19, -6

Thus the statement will be executed 11 times with the control variable COUNT taking each of the values 1,7,8,10,11,12,13,14,15,19,-6 in turn.

Labels which occur inside a compound statement controlled by a FOR clause should not be used outside that compound statement. If a GO TO statement outside the compound statement causes a branch to a label inside the

compound statement then the initialisation of the control variable will be by-passed and the FOR statement will probably not behave in the manner intended.

When a FOR statement terminates because the control variable has passed its limiting value the control variable retains the value it has after passing the limit.

Examples

```
1.  FOR I := 0 STEP 2 UNTIL 6 DO A := A + I;
```

The statement A := A + I is executed 4 times with I taking each of the values 0, 2, 4 and 6 in turn. After the last execution I is again incremented by a step of 2 to hold the value 8 and is then past the limit 6. Hence the value retained by I on exit from the FOR statement is 8.

```
2.  FOR I := 1 STEP 1 UNTIL 3 DO
    BEGIN
      A := A - I;
      IF A = 0 THEN I := 4
    END;
```

Suppose that A is an integer variable and holds the value 3 initially. The compound statement will be executed twice with I having the values 1 and 2 in that order but on the second iteration I will be assigned the value 4. At the end of the second iteration I will again be incremented by a step of 1 to hold 5 and will be tested against the limiting value 3. Hence the FOR statement will terminate and I will retain the value 5.

If a FOR statement terminates because of a GO TO branching out of the FOR statement, the control variable retains its current value.

Example

```
    FOR I := 1 STEP 1 UNTIL 28 DO
    BEGIN
       A := A * I;
       IF A > 25 THEN GO TO L
    END;
L:  A := A + I;
```

In this example if A is an integer variable initialised to 1 then the condition A > 25 will be satisfied on the fifth iteration when I = 5. The GO TO L instruction causes a branch out of the FOR statement and leaves I holding the value 5.

2.8 FOR-WHILE statements

There are 2 forms of the FOR-WHILE statement:

1. FOR control variable := AE STEP AE WHILE BE DO statement;
2. FOR control variable := AE WHILE BE DO statement;

Where 'AE' represents an arithmetic expression and 'BE' represents a Boolean expression.

In both cases the 'statement' is executed repeatedly until the Boolean expression is false. When more than one statement is to be executed iteratively a compound statement must be used to group them together.

Examples

1. FOR I := 1 STEP 1 WHILE SUM < 10 DO SUM := SUM + I;

The statement SUM := SUM + I is executed repeatedly while the Boolean expression SUM < 10 is true. The statement is first executed with I = 1 and for each subsequent execution the value of I is incremented by 1. In this example therefore, if SUM is an integer variable which is initially zero, the statement will be executed 4 times with I taking the values 1, 2, 3, 4 in turn and leaving SUM holding the value 10. At this point the Boolean expression SUM < 10 will be false.

2. FOR I := 1 WHILE SUM < 10 DO SUM := SUM + I;

This is very similar to the last example except that this time I = 1 at every execution of the statement SUM := SUM + I. Thus SUM is incremented 10 times altogether and its final value is 10.

The difference between the FOR-WHILE statement and the ordinary FOR statement is that in a FOR-WHILE statement the iteration is controlled by a Boolean expression which can be entirely independent of the control variable, whereas in the ordinary FOR statement the iteration is controlled by the set of possible values which the control variable may take.

Boolean variables may serve as the Boolean expressions in FOR-WHILE statements.

Example If UNFINISHED is a Boolean variable then it may be used to control an iteration as follows:

```
UNFINISHED := TRUE;
FOR K := 1 STEP 1 WHILE UNFINISHED DO
BEGIN
   READ (CARD, /, A, B);
   IF A = 0 AND B = 0 THEN UNFINISHED := FALSE ELSE
   BEGIN
      MEAN := (A + B)/2;
      WRITE (LP, / K, MEAN)
   END
END;
```

With this piece of code, pairs of numbers A, B are repeatedly read in from the card reader file, CARD, and processed until a pair of zeros is encountered after which the iteration terminates. When A and B are not both zero their mean is calculated and output along with the pair number for identification purposes. Notice that the FOR-WHILE clause operates on a compound statement which groups the necessary statements together.

A FOR statement may include WHILE clauses as well as other types of for-lists.

Example

```
FOR I := 2, 7 STEP 2 WHILE X < 0, 10 STEP -1 UNTIL 3,
         6, -6 WHILE X < 10 DO statement;
```

The 'statement' here will be executed with I taking the values

1) 2,
2) 7,9,11, ... while X < 0
3) 10,9,8,7,6,5,4,3,
4) 6,
5) -6 repeatedly while X < 10

A program can easily get stuck in an infinite loop if the programmer is not careful.

Example To find the value of factorial N where N is the smallest integer such that factorial N > 232 the following piece of program might be used:

```
PROD := 1;
FOR N := 1 STEP 1 WHILE PROD LEQ 232 DO PROD := PROD * N;
WRITE (LP, /, PROD);
```

If "STEP 1" is accidently omitted however, the code becomes

```
PROD := 1;
FOR N := 1 WHILE PROD LEQ 232 DO PROD := PROD * N;
WRITE (LP, /, PROD);
```

in which PROD permanently holds the value 1 and PROD LEQ 232 is always true. Consequently, at execution time the program will go into an infinite loop, re-executing the FOR-WHILE statement indefinitely.

2.9 THRU statements

The general form of a THRU statement is

```
THRU AE DO statement;
```

where 'AE' is an arithmetic expression. The arithmetic expression is evaluated and converted to an integer by rounding if real, and the 'statement' is executed the number of times indicated by the result. A compound statement must be used if more than one statement is to be executed iteratively.

Example To read in 10 pairs of numbers and output their means the following code might be used:

```
THRU 10 DO
BEGIN
   READ (CARD, /, A, B);
   MEAN := (A + B)/2;
   WRITE (LP, /, MEAN)
END;
```

In this example the whole of the compound statement will be executed 10 times altogether.

The THRU statement is similar to a FOR statement with a STEP size of 1 except that the former has no control variable. Indeed, any iteration controlled by a THRU statement can be rewritten using a FOR statement in which the control variable is used only as a counter but in such situations it is probably easier to use a THRU statement.

2.10 WHILE Statements

The general form of a WHILE statement is

WHILE Boolean expression DO statement;

The 'statement' is executed repeatedly until the 'Boolean expression' is false. If the expression is initially false then the 'statement' is never executed.

Examples

1. WHILE SUM < 10 DO SUM := SUM + 4;

If SUM = 0 initially and is an integer variable then the statement SUM := SUM + 4 is executed 3 times. After the third execution SUM = 12 and the iteration is terminated since the Boolean expression SUM < 10 is no longer true.

2. A program which reads in pairs of numbers, calculates and prints their mean and then recycles until a pair of zeros is encountered might use a WHILE statement as follows:

```
READ (CARD, /, A, B);
WHILE NOT (A = 0 AND B = 0) DO
BEGIN
   MEAN := (A + B)/2;
   WRITE (LP, /, MEAN);
   READ (CARD, /, A, B)
END;
```

Notice that if the first pair of numbers are both zero then no output is obtained and the iteration stops at once. In this case the WHILE clause operates on a compound statement which is used to group the 3 simple statements together so that all 3 are executed each time.

3. A Boolean variable may be used to control the iteration:

```
OK := TRUE;
WHILE OK DO
BEGIN
   A := A - 1;
   OK := A NEQ 0;
   IF OK THEN
   BEGIN
      B := B/A;
      OK := B > A
   END
END;
```

Here assignments are made to A and B until either A = 0 or B $\leq$ A. Observe that OK is also used within the compound statement to prevent a possible division by zero.

The difference between a WHILE and a FOR-WHILE statement is that no control variable is used with a WHILE statement. If a control variable is required then a FOR-WHILE statement should be used.

2.11 DO-UNTIL statements

The general form of a DO-UNTIL statement is

DO statement UNTIL Boolean expression;

The 'statement' is executed repeatedly until the 'Boolean expression' is true. Using this construction the 'statement' is always executed at least once and this is what distinguishes it from a WHILE statement in which the 'statement' may not be executed at all. Note that there is no semicolon between the 'statement' and the UNTIL.

Examples

1.
```
DO SUM := SUM + 4 UNTIL SUM > 10;
```

If SUM is an integer variable and is initially 0 then SUM will be incremented by 4, 3 times after which SUM = 12 and the Boolean expression SUM > 10 is true. If SUM = 15 initially it will be incremented by 4 just once after which SUM = 19 which satisfies the Boolean expression.

2.
```
DO
BEGIN
   READ (CARD, /, A, B);
   MEAN := (A + B)/2;
   WRITE (LP, /, MEAN)
END
UNTIL A = 0 AND B = 0;
```

In this example pairs of numbers are read in and processed until 2 zeros are encountered at which point the iteration stops. Notice that when 2 zeros are encountered their mean will still be output. A compound statement is necessary in this example since all 3 simple statements have to be executed on each cycle.

2.12 Nested FOR Statements

Consider a program which reads in 5 pairs of numbers and prints their maximum together with an identifying pair number. The relevant piece of program might make use of a FOR statement as follows:

```
FOR PAIRNO := 1 STEP 1 UNTIL 5 DO
BEGIN
   READ (CARD, /, X, Y);
   IF Y > X THEN MAX := Y ELSE MAX := X;
   WRITE (LP, /, PAIRNO, MAX)
END;
```

Notice that this piece of code is of the form:

```
FOR CV := for-list DO statement;
```

where the 'statement' is a compound statement. Thus the above code is nothing more than a FOR statement.

Now suppose that instead of just 1 set of 5 pairs of numbers there are 10 such sets (i.e. 50 pairs altogether in 10 sets of 5). It may now be required to output the maximum value of each pair together with identifying set and pair numbers. It is therefore required to perform the above FOR statement 10 times altogether making a slight alteration to the WRITE statement so that the set number is also printed. This may be achieved using another FOR statement which operates on the whole of the above FOR statement:

```
FOR SETNO := 1 STEP 1 UNTIL 10 DO
FOR PAIRNO := 1 STEP 1 UNTIL 5 DO
BEGIN
   READ (CARD, /, X, Y );
   IF Y > X THEN MAX := Y ELSE MAX := X;
   WRITE (LP, /, SETNO, PAIRNO, MAX)
END;
```

Observe that this piece of code is of the form:

```
FOR SETNO := 1 STEP 1 UNTIL 10 DO statement;
```

where the 'statement' is itself a FOR statement. Thus the inner FOR statement will be executed 10 times. Since the compound statement is executed 5 times for each execution of the inner FOR statement it follows that the compound statement will be executed 10 x 5 = 50 times altogether as required.

When one FOR statement is inside another like this, they are said to be 'nested'. This construction of nested FOR statements is one which is frequently used in ALGOL particularly in connection with array manipulation as can be seen in the next chapter. Note that the programmer is not restricted to a nesting depth of two but may have many FOR statements within other FOR statements.

2.13 Comments

B6700 ALGOL allows the programmer to include 'comments' or explanatory text in programs. These comments are entirely ignored by the compiler but can help other readers to understand the program. Comments must adhere to the following rules:

1. A comment following a BEGIN, a declaration or a statement may consist of the word COMMENT followed by a space and any sequence of characters excluding a semicolon. A semicolon is used to terminate the comment.

2. A comment following an END may consist of any sequence of letters, digits and spaces with the exception of END, UNTIL, ELSE.

3. A comment may also be introduced by a % sign followed by the comment. The end of the card is taken as the terminator of the comment.

Example The following program shows some situations in which comments may occur:

```
BEGIN
   COMMENT THIS IS A COMMENT DEMONSTRATION PROGRAM;
   % SO FAR NOTHING HAS BEEN DONE.
   INTEGER I; % I IS AN INTEGER
   REAL J;
   COMMENT J IS A REAL NUMBER AND THIS IS A COMMENT;
   I := 0;
   COMMENT THE LAST LINE IS A STATEMENT;
   J := 1
   COMMENT THERE IS A NO SEMI-COLON AFTER THE LAST STATEMENT;
END OF A VERY SILLY PROGRAM.
```

CHAPTER 3

ARRAYS

3.1 Introduction

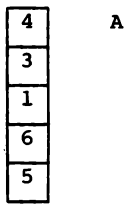
Consider a program which reads in 5 numbers, outputs their mean and lists the difference between each number and the mean.

Firstly, all the numbers are read in and summed.
Secondly, the mean is calculated and output.
Thirdly, the mean is compared with each of the 5 numbers.

It is in step 3 that problems occur for with the techniques already described either all of the 5 numbers must be re-read (usually impossible) or 5 identifiers must be declared in which each of the 5 numbers can be stored and re-accessed using 5 statements. Now while the second alternative may seem reasonable with 5 numbers it is clearly unacceptable when there are 500. For this reason ALGOL provides arrays (like tables) to ease the problem whereby a whole set of numbers can be stored together under one identifier and accessed in one statement.

3.2 One-dimensional Arrays

A one-dimensional array (sometimes called a vector) may be visualised as a one-dimensional table (or list) of numbers as shown below; the whole table is given a name and each element or box in the table is identified by means of a 'subscript'.



In this example, if the boxes in A are numbered from 1 to 5 then A[1] denotes the contents of the first box in A, i.e. A[1] = 4. Here A is the array's identifier and 1 is the subscript by which the first element of A is identified. In ALGOL, subscripts are contained in square brackets [ ].

Similarly A[3] = 1 i.e. the contents of the third element of the array A, and A[5] = 5. Although in this case each element of A contains a different value, there is no reason why several or all of its elements should not have the same value.

One-dimensional arrays may have any number of elements but would normally have more than one.

3.3 Two-dimensional Arrays

A two-dimensional array may be visualised as a two-dimensional table of numbers as shown below. This time two subscripts are required to identify an element in the table. The first subscript indicates the row containing

the required element and the second indicates the column. A comma is used to separate the subscripts.

		columns 1	2	3	4
rows	1	90	89	80	53
	2	72	76	78	71
	3	49	10	56	60
	4	60	58	88	76
	5	25	40	38	52

MARKS

A two-dimensional array such as MARKS here, might be used to contain the marks obtained by 5 students in 4 examinations. Each row contains the marks of 1 student and each column contains the marks obtained in a particular examination. Thus

MARKS [3, 2] = 10 i.e. the contents of the element in the third row and second column.

MARKS [5, 4] = 52 i.e. the contents of the element in the fifth row and fourth column.

Note that MARKS [4, 5] does not exist since there is no column 5.

MARKS is said to be a 5 x 4 array since it has 5 rows and 4 columns. Two-dimensional arrays may have any number of rows and columns but would normally have more than one of each.

3.4 Array Declarations

Arrays must be declared giving their type, number of dimensions and the number of elements in them. The general form of an array declaration is

```
type ARRAY arrayname [ bound pair list ];
```

The 'type' can be REAL, INTEGER or BOOLEAN but if it is not specified then REAL will be assumed. The 'arrayname' can be any identifier chosen by the programmer and the 'bound pair list' is a list of 'bound pairs' separated by commas with one 'bound pair' for each dimension of the array. A bound pair is of the form

```
lower bound     upper bound
```

where the lower and upper bounds specify the range of the dimension in question.

Examples

1. If the array A of section 3.2 is to hold integer values only throughout the program then the appropriate declaration for A is

```
INTEGER ARRAY A[1 : 5];
```

The bound pair list consists of only one bound pair since A is one-dimensional. The bound pair 1 : 5 indicates that A is to have 5 elements and that these have to be numbered from 1 to 5. Thus A has elements A[1], A[2], A[3], A[4], A[5]. The lower bound does not have to be 1 as in this case and it is frequently preferable to have a lower bound of 0. If the declaration

```
INTEGER ARRAY A[0 : 4];
```

were made instead, then A would still have 5 elements but this time they would have been numbered from 0 to 4. Thus A would consist of the elements A[0], A[1], A[2], A[3] and A[4]. The array A may also have negative subscripts if this is desirable. For instance,

```
INTEGER ARRAY A[-2 : 2];
```

has elements A[-2], A[-1], A[0], A[1], A[2] and

```
INTEGER ARRAY A[-5 : -1];
```

has elements A[-5], A[-4], A[-3], A[-2] and A[-1]. Note that the lower bound must never be greater than the upper bound, i.e. -1 : -5 is not a valid bound pair.

2. If the 5 x 4 array B above, is to take real values then its declaration would be

```
REAL ARRAY B[1 : 5, 1 : 4];  or ARRAY B[1 : 5, 1 : 4];
```

Thus B is a two-dimensional array since it has 2 bound pairs. The first dimension having a range of 1 to 5 and the second a range of 1 to 4. As in the previous example zero or negative subscripts may be used if these are preferable.

3. Arrays are not restricted to just one or two dimensions but may have any number. A 4-dimensional array C might be declared as

```
INTEGER ARRAY C[1 : 5, 1 : 2, 3 : 6, -1 : 2];
```

Altogether C has 5 x 2 x 4 x 4 = 160 elements.

4. Several arrays of the same type can be declared together by separating them with commas. Hence 2 real arrays B and D might be declared

```
REAL ARRAY B[1 : 5, 1 : 4], D[0 : 3];
```

Here B is a 5 x 4 real array and D is a 1-dimensional real array with 4 elements. If several arrays have the same type and the same number of dimensions over the same ranges then they may be declared together without giving the bound pair list in each case. For example,

```
REAL ARRAY F, G[1 : 6], H, X, Y[1 : 60, 2 : 10];
```

declares 5 real arrays of which F and G are 1-dimensional and H, X and Y are 2-dimensional. Both F and G have 6 elements numbered from 1 to 6 and H, X and Y are all 60 x 9 arrays where the first dimension ranges from 1 to 60 and the second from 2 to 10.

3.5 Subscript Expressions

Array elements such as A[4] and B[5, 4] are known as 'subscripted variables' as distinct from 'simple variables' such as MEAN, I etc. Until now every subscript of all the subscripted variables has been a number but subscripts may in fact be arithmetic expressions.

Examples

1. The subscript of A[I + 1] is the arithmetic expression I + 1 which is clearly dependent on the value of I. If I = 2 then A[I + 1] is equivalent to A[3].

2. If I = 2 and J = 3 then C[I, J, I + J, I - J] is equivalent to C[2, 3, 5, -1].

3. If A[2] = 3 then B[4, A[2]] is equivalent to B[4, 3]. The second subscript is A[2].

4. Assuming still that A[2] = 3 then A[A[2]] is equivalent to A[3]. Again the subscript is A[2].

5. If I = 2, J = 3 and A[2] = 3 then A[A[I] + J] is equivalent to A[6].

The value of a subscript must not lie outside the range specified in the bound pair list in the declaration. For example if A is declared as

```
INTEGER ARRAY A[0 : 3];
```

the element A[4] does not exist and cannot be accessed. Hence if I = 2 then trying to use A[I + 2] will cause a failure. Similarly it would be impossible to access A[-1].

3.6 Application of Arrays

It should now be possible to write a program which reads in 500 numbers, say, and outputs their mean together with the difference between each of the 500 numbers and the mean. Suppose initially that each of 500 numbers is on a separate card then the necessary steps to be performed are

1. Read in 500 numbers one at a time, each from a different card, storing each number in an array and accumulating their sum,

2. Divide the total by 500 to obtain the mean,

3. Write out the mean,

4. Calculate and print the difference between each of the 500 numbers and the mean.

The following program achieves this.

```
BEGIN
   FILE CARD(KIND=READER), LP(KIND=PRINTER);
   REAL ARRAY NUMBER[1 : 500];
   REAL MEAN, SUM, DIFFERENCE;
   INTEGER I;
   SUM := 0.0;
   FOR I := 1 STEP 1 UNTIL 500 DO
   BEGIN
      READ(CARD, /, NUMBER[I]);
      SUM := SUM + NUMBER[I]
   END;
   MEAN := SUM/500;
   WRITE(LP, /, MEAN);
   FOR I := 1 STEP 1 UNTIL 500 DO
   BEGIN
      DIFFERENCE := MEAN - NUMBER[I];
      WRITE(LP, /, DIFFERENCE)
   END
END OF PROGRAM 2.
```

Program 2

Note that once NUMBER has been initialised by the READ statement the data is stored there permanently; it is not necessary to read the data in again when NUMBER is next accessed. If it is required, of course, the values held in an array can be altered just like any other variables but in program 2 the array NUMBER is not altered after initialisation and therefore holds the same values throughout the life of the program.

Now clearly it is very wasteful and inconvenient to have to put only one number on a card as is required by program 2 but fortunately an improved form of array input/output is available.

3.7 Array Input/Output

In I/O statements there is a special notation called an 'array row' which can be used to denote the whole of a 1-dimensional array or the whole of one row of an array with more than 1 dimension.

Examples

1. The 'array row' NUMBER[*] denotes the whole of the 1-dimensional array NUMBER.

2. The 'array row' B[2, *] denotes the whole of the second row of the 2-dimensional array B.

Only one asterisk may be used in an array row so that B[*,*] may not be used to denote the whole of the 2-dimensional array B. Furthermore, the asterisk in an array row must come to the right of any other subscripts. Hence while B[2, *] is valid, B[*,2] is invalid and therefore cannot be used to denote the whole of column 2 of B. Array rows are not restricted to 1- or 2-dimensional arrays however and C[1, 2, 2, *] is a legitimate array row of a 4-dimensional array C if the subscripts 1, 2, 2 are valid.

The whole of an array row may be read or written using the following

construction:

```
READ
      (filename, format, array row);
WRITE
```

Examples

1. To read in the whole of the 1-dimensional array NUMBER the array row NUMBER[*] may be used:

```
READ(CARD, /, NUMBER[*]);
```

2. To write out the whole of row 3 of the 2-dimensional array B the array row B[3, *] may be used:

```
WRITE(LP, /, B[3, *]);
```

The READ statement in example 1 causes enough numbers to be read in to completely fill NUMBER. It does not matter how these numbers appear on the data cards just so long as they are separated by commas and no number is split up so that part is on one card and part on the next. Thus each card can have as many numbers as can fit on it instead of just one. The WRITE statement in example 2 causes numbers to be written across the page instead of just one number per line.

Using the array row notation it is easy to print a 2-dimensional array so that each row is on a separate line. Remembering that each WRITE statement starts on a new line this objective can be achieved by using a WRITE statement controlled by a FOR clause.

Example If B is an R x C 2-dimensional array and the rows are numbered from 1 upwards, then the statement

```
FOR I:= 1 STEP 1 UNTIL R DO WRITE(LP,/, B[I, *]);
```

which is equivalent to R WRITE statements, will print each row on a new line.

Similar techniques can be applied to read in one row at a time from cards.

Program 2 of section 3.6 can now be rewritten making more efficient use of materials and one possibility is the following program:

```
BEGIN
   FILE CARD(KIND=READER);
   FILE LP(KIND=PRINTER);
   INTEGER I;
   REAL SUM, MEAN;
   REAL ARRAY NUMBER, DIFFERENCE[1 : 500];
   READ(CARD, /, NUMBER[*]);
   SUM := 0.0;
   FOR I := 1 STEP 1 UNTIL 500 DO
   SUM := SUM + NUMBER[I];
   MEAN := SUM/500;
   FOR I := 1 STEP 1 UNTIL 500 DO
   DIFFERENCE[I] := MEAN - NUMBER[I];
   WRITE(LP, /, MEAN, DIFFERENCE[*])
END OF PROGRAM 3.
```

When the array elements to be read/written do not conveniently form an array row, the array row notation cannot be used and a method is required which will indicate precisely which elements of the array are to be accessed. For example, it may be required to read 5 numbers into the array NUMBER starting at NUMBER[20]. This could be done using the statement:

```
READ(CARD, /, NUMBER[20], NUMBER[21], NUMBER[22],
              NUMBER[23], NUMBER[24]);
```

but this involves writing out a long and tedious list. Instead the list can be abbreviated using a FOR clause:

```
READ(CARD, /, FOR I := 20 STEP 1 UNTIL 24 DO NUMBER[I]);
```

which has exactly the same effect as the statement above but is much more manageable.

In the same way if PEN is a 2-dimensional array

```
READ(CARD, /, FOR J := 2 STEP 1 UNTIL 20 DO PEN[6, J]);
```

will read values into PEN[6, 2], PEN[6, 3], ..., PEN[6, 20].

A similar construction can be used to write out the first 10 numbers of the array NUMBER:

```
WRITE(LP, /, FOR I := 1 STEP 1 UNTIL 10 DO NUMBER[I]);
```

Note that this is just one WRITE statement and therefore the 10 numbers will all appear on the same line. This technique may be extended to print the first 30 elements, say, of NUMBER with 10 numbers on each line:

```
FOR J := 0 STEP 10 UNTIL 20 DO
WRITE(LP, /, FOR I := 1 STEP 1 UNTIL 10 DO NUMBER[I + J]);
```

Here the WRITE statement is executed 3 times with J taking each of the values 0, 10, 20 in turn. Each WRITE takes a new line and then prints 10 numbers.

It may be convenient to output more than one array at a time and this too may easily be achieved.

Examples

1. If A is a 1-dimensional array with R elements and B is a 2-dimensional array with R rows then the instruction

```
FOR I := 1 STEP 1 UNTIL R DO
WRITE(LP, /, B[I, *], A[I]);
```

will output A and B with one row of B followed by one element of A on each line.

2. If A and D are both 1-dimensional arrays with R elements then the statement

```
WRITE(LP, /, FOR I := 1 STEP 1 UNTIL R DO [A[I], D[I]]);
```

will cause the elements of A and D to be printed across the page in the form A[1], D[1], A[2], D[2], ... A[R], D[R].

Note that in READ/WRITE statements where more than one list element is controlled by one FOR clause the appropriate list elements are enclosed in square brackets. Hence in the WRITE statement in the last example the elements A[I], D[I] are enclosed in square brackets since both are to be controlled by the FOR clause.

3.8 Array Manipulation

Most programs involving arrays call for some sort of array manipulation at some stage of the program. For instance it is not uncommon to add or multiply two arrays together, sort an array into ascending or descending order or search through an array looking for zeros etc. These sorts of operations are most easily done with FOR statements which must frequently be nested. The following examples look at some typical problems and the way in which they might be tackled.

Example 1. Add together 2 1-dimensional arrays A and B where each consists of 36 elements numbered from 1 to 36, and leave the result in A. This could be achieved with the 36 assignment statements

```
A[1] := A[1] + B[1];
A[2] := A[2] + B[2]; etc.
```

but is more easily achieved using the FOR statement:

```
FOR I := 1 STEP 1 UNTIL 36 DO A[I] := A[I] + B[I];
```

which has the same effect.

Example 2. Count the number of zeros in row 3 of the 2-dimensional integer array SCORE which has 25 elements numbered from 0 to 24 in each row. Leave the result in an integer variable COUNT.

```
COUNT := 0;
FOR J := 0 STEP 1 UNTIL 24 DO
IF SCORE[3, J] = 0 THEN COUNT := COUNT + 1;
```

Here the IF statement being controlled by the FOR statement is executed 25 times and accesses a different element in row 3 of SCORE each time.

Example 3. If the array SCORE in the last example has 5 rows numbered from 1 to 5, count how many of the 125 elements in SCORE are zero. This can be done by extending the code in example 2 so that all the rows of SCORE are examined instead of just row 3.

```
COUNT := 0;
FOR I := 1 STEP 1 UNTIL 5 DO
FOR J := 0 STEP 1 UNTIL 24 DO
IF SCORE[I, J] = 0 THEN COUNT := COUNT + 1;
```

This time the inner FOR statement is executed 5 times with I taking the values 1, 2, 3, 4 and 5 in turn. Since the inner FOR statement causes the Ith row to be scanned for zeros it follows that these instructions cause each of the rows 1, 2, 3, 4 and 5 to be scanned as required.

Example 4. POINTS is a 26 x 8 real array in which the rows are numbered from 1 to 26 and the columns from 1 to 8. Fill POINTS with the value 5.1. In this case it makes no difference whether POINTS is filled row by row or column by column.

```
FOR I := 1 STEP 1 UNTIL 26 DO
FOR J := 1 STEP 1 UNTIL 8 DO POINTS[I, J] := 5.1;
```

With the FOR statements in this order POINTS is filled row by row since for each value of I (i.e. the row subscript) the column subscript J takes all possible values. By reversing the order of the FOR statements POINTS will be filled column by column:

```
FOR J := 1 STEP 1 UNTIL 8 DO
FOR I := 1 STEP 1 UNTIL 26 DO POINTS[I, J] := 5.1;
```

Example 5. A, B and C are all 5 x 10 x 16 arrays in which each dimension is numbered from 0 upwards. Add together A and B placing the result in C.

```
FOR H := 0 STEP 1 UNTIL 4 DO
FOR I := 0 STEP 1 UNTIL 9 DO
FOR J := 0 STEP 1 UNTIL 15 DO
C[H, I, J] := A[H, I, J] + B[H, I, J];
```

These 3 nested FOR statements (one for each dimension) cause the assignment statement to be executed 5 x 10 x 16 = 800 times altogether with a different permutation of H, I and J each time.

Example 6. A and B are both 9 x 9 arrays in which the rows and columns are numbered from 1 to 9. Place the transpose of B in A so that column 1 of B forms row 1 of A, column 2 of B forms row 2 of A etc.

```
FOR I := 1 STEP 1 UNTIL 9 DO
FOR J := 1 STEP 1 UNTIL 9 DO
A[I, J] := B[J, I];
```

Note that A is filled row by row and B is read column by column thus obtaining the required result.

CHAPTER 4

MORE ADVANCED INPUT/OUTPUT

4.1 Introduction

Until now the general form of READ/WRITE statements used has been

```
READ
       (filename, /, list of variables);
WRITE
```

A more general, but more complicated form is

```
READ
       (file part, format, list);
WRITE
```

where the 'file part' consists of the file name to be accessed but may be followed by an optional part in square brackets; the 'format' may take many forms rather than simply '/' which is only appropriate for free-field data; and the 'list' may include arithmetic expressions as well as variables.

4.2 The READ Statement

[SPACE n] placed after the filename will cause n cards to be skipped over without being read, before the read is executed.

[NO] placed after the filename will cause the next READ statement to access the same card as the current READ statement.

Examples 1.

```
READ (CARD [NO], /, A);
READ (CARD, /, B);
```

This will read the same value into B as was read into A because the second READ statement accesses the same card as the first READ statement.

2.

```
READ (CARD, /, A);
READ (CARD [SPACE 2], /, B);
```

If A is read from card 1 then B will be read from card 4 since the second and third cards will be skipped over.

Note that there must be no comma between the filename and the square bracket when present.

When the data is in free-field form a format part consisting of '/' is adequate. If this is not the case then a format may be used which consists of several editing phrases enclosed in broken brackets < >. These editing phrases enable the programmer to define the position of data on the card or line and the way in which it is to be interpreted. The meaning of some of the more useful editing phrases will now be described.

Fm.n means that a number is to be read from the next m columns of the current card and is to be interpreted as a decimal number having n digits after the decimal point. If the number read from the card contains an explicit decimal point then this overrides the implied decimal point in the format specification. Blanks are interpreted as zeros.

Example If a READ statement is reading in a value at the start of a card and the format specified is F6.2 then the table below shows how the number will be interpreted:

CARD COLUMNS 1	2	3	4	5	6	INTERPRETED AS
		1	2	8	3	12.83
	-	1	2	8	3	-12.83
-	1		2	8	3	-1.283
					1	0.01
+	3	0	4			304.00
1	2	3	4	5	6	1234.56
	2	1				210.00
	2	1		3	2	210.32

Im means that a number is to be read from the next m columns of the current card and is to be interpreted as an integer. Again blanks are interpreted as zeros.

Xm means that the next m columns of the current card are to be skipped over and ignored.

/ means that the rest of the current card is to be ignored and the next card taken.

A repeat factor may be used in the editing specifications to indicate that part of the editing phrases are to be used more than once.

Example 3F6.2 is equivalent to F6.2, F6.2, F6.2

2I5 is equivalent to I5, I5

2(F6.2, 2I5) is equivalent to F6.2, I5, I5, F6.2, I5, I5.

With the exception of X and /, each occurrence of the editing phrases has a corresponding list element. Each list element is paired with an editing phrase so that ignoring each X and / and going from left to right, the nth editing phrase corresponds to the nth list element. If there are more editing phrases than list elements then the extra phrases are ignored but if there are fewer then the editing phrases are re-used until the list is satisfied. Each time the editing phrases are re-used a new card is taken also.

Examples

1. The values of 2 integers I and J are to be read in from columns 25 - 29 of the first card and columns 2 - 6 of the second. The following instruction achieves this.

```
READ(CARD, < X24, I5, /, X1, I5 > , I, J);
```

Note that the editing phrases X24, / and X1 have no list element corresponding to them. The list element I is paired with the first I5 and J is paired with the second.

2. 30 6-digit numbers are to be read from cards into elements 1 to 30 of a 1-dimensional array A where each number is separated from the next by 4 spaces and is to be interpreted as having 2 digits after the decimal point. This can be done using the READ statement:

```
READ(CARD, < 8(F6.2, X4) >, FOR I := 1 STEP 1 UNTIL 30 DO A [I]);
```

Because each number being read in contains 6 digits and is followed by 4 spaces each card contains precisely 8 numbers. Thus after reading in 8 numbers a new card has to be taken. Since the format does not contain enough editing phrases for all 30 list elements it is re-used as necessary i.e. after every eighth number, and therefore the next card is also taken at the same time. The extra editing phrases at the end are ignored.

Note that if by accident there are too few editing phrases for the number of list elements, a new card will be taken where it was not intended to do so and the program will misbehave badly.

4.3 The WRITE Statement

[SPACE n] placed after the filename will cause the line printer to advance the paper by n lines after the current line has been printed. When the next WRITE statement is executed the data will be printed without any further paper motion and consequently there will be only n - 1 blank lines between the two lines of print. Therefore to obtain b blank lines after the current line of output [SPACE b + 1] should follow the filename in the WRITE statement.

[SKIP 1] placed after the filename will cause a new page to be started at the line printer after the current line has been printed.

Examples

1.
```
WRITE(LP [SPACE 3], /, A);
WRITE(LP, /, B);
```

A will be printed, then there will be 2 blank lines and then B will be printed.

2.
```
WRITE(LP [SKIP 1], /, A);
WRITE(LP, /, B);
```

A will be printed and then B will be printed at the top of a new page.

The meaning of some of the more useful editing phrases which may be used in the format part of a WRITE statement will now be described.

Fm.n means that the value of the corresponding list element is to be written in the next m columns of the current line giving n digits after the decimal point. One column is always required for the decimal point and in the case of negative numbers one column is also required for the sign. Thus a negative number having up to m-n-2 digits before the decimal point may be written out and a non-negative number having up to m-n-1 digits before the point may be written out. Leading zeros are suppressed and the number is rounded in the nth decimal place. Attempts to write out a value which does not fit in the given format will result in m asterisks being output but the program will continue.

Im means that the value of the corresponding list element is to be written in the next m columns of the current line as an integer. As before, one column is required for the sign if the number is negative. If the corresponding list element is a real value it will be rounded to the nearest integer before being output.

Xm means that the next m columns are to be left blank.

/ means that a new line is to be taken at the line printer.

Explanatory text for headings etc. may be output by enclosing the required text in quotation marks (") and including it as part of the format.

Repeat factors may be used as detailed in the previous section.

Examples

1. `WRITE(LP, < "MEAN =", F6.2 >, MEAN);`

this will write out the quotation MEAN = followed by the value of MEAN to 2 decimal places allowing 2 or 3 digits before the decimal point depending on whether MEAN is negative or non-negative respectively. For instance if the value of MEAN is -2.835 then the output would be

```
MEAN = -2.84
```

and would appear in the first 12 columns of the line.

2. To write out the mean and sum of 2 integers A and B on separate lines the following WRITE statement might be used:

```
WRITE(LP, < "MEAN =", F6.1, /, "SUM =", I6 >, (A+B)/2, A+B);
```

If there are more editing phrases than list elements then the extra editing phrases are ignored but if there are fewer the editing phrases are re-used until the list is satisfied. Each time they are re-used a new line is started at the line printer.

4.4 Format Designators and Lists

It is possible that in one program the same format is used several times in different READ or WRITE statements. Similarly the same list of variables may occur several times also. To save writing out each format and list explicitly, format and list identifiers may be used but as always must be declared first.

The general form of a format declaration is

```
FORMAT identifier (format);
```

where the 'identifier' can be any suitable name chosen by the programmer using the usual rules for identifiers. The 'format' is the list of editing specifications that would normally appear in the READ/WRITE statement but no broken brackets are required.

Example

```
FORMAT F1 (X5, "TIME =", F6.2, /, "SPEED =", F6.2);
```

The general form of a list declaration is

```
LIST identifier (list of variables);
```

The 'identifier' is chosen by the programmer and follows the usual rules for identifiers. The 'list of variables' is that list of variables which are to be read/written. The variables in the list are separated by commas and must all have been previously declared.

Example

```
LIST L1 (T2, S1);
LIST L2 (FOR I := 1, 2, 3 DO [A[I], B[I]], C);
```

A READ or WRITE statement may contain format and list identifiers instead of actual formats and lists.

Examples

1.
```
READ(CARD, < 2F10.6 >, L1);
```

is equivalent to

```
READ(CARD, < 2F10.6 >, T2, S1);
```

2.
```
WRITE(LP, F1, L1);
```

is equivalent to

```
WRITE(LP, < X5, "TIME=", F6.2, /, "SPEED=", F6.2 >, T2, S1);
```

3.
```
WRITE(LP, /, L2);
```

is equivalent to

4.5 Action Labels

Action labels provide a means of transferring control from READ and WRITE statements when exception conditions occur - that is to say when something goes wrong. Up to three labels may be used and these are separated by colons and enclosed in square brackets.

Example

```
READ(CARD, < 2F6.2, I5 >, A, B, C)  [L1 : L2 : L3];
```

The action labels here are L1, L2 and L3.

Consider the general case where there are 3 labels given - LABEL 1, LABEL 2 and LABEL 3 in that order. A branch to LABEL 1 takes place when an 'end-of-file' condition occurs; i.e. an attempt is made to read or write and there is nothing to read from or write onto. This condition usually arises because there are no data cards (or not enough) included with a program which attempts to read in values of its variables.

A branch to LABEL 2 takes place when an irrecoverable parity error occurs. Parity errors may not be the fault of the programmer and need not concern the beginner unduly.

A branch to LABEL 3 takes place if there is a conflict between the format and the data. Such a conflict may arise for example, if an attempt is made to read alphabetic characters into a real variable using an Fm.n format. These errors are usually the result of an oversight on the part of the programmer or the result of a punching error.

Hence a programmer might use 3 action labels EOF, PARITY and CONFLICT, say, corresponding to each of the above 3 possible errors. In this case the action labels would be written as [EOF : PARITY : CONFLICT]. Any selection of the 3 action labels may be used but a problem of notation arises when only 1 or 2 labels are being used. For instance if only one label, L, is specified it is unreasonable to expect the computer to know on which of the 3 possible error conditions a branch to label L has to be made. Colons are therefore used to clarify matters and the following table shows how they are arranged to ensure that the given labels are utilised for the appropriate error conditions.

1 label used	2 labels used
[EOF]	[EOF : PARITY]
[: PARITY]	[EOF :: CONFLICT]
[:: CONFLICT]	[: PARITY : CONFLICT]

Exception conditions occurring during a READ or WRITE statement may be handled without the use of action labels. READ and WRITE statements return the Boolean value TRUE when something goes wrong such as when the end-of-file condition is raised. Thus it is possible to write:

```
WHILE NOT READ(CARD, /, A, B) DO SUM := SUM + A + B;
```

This instructs the computer to read in pairs of numbers, accumulating their totals in SUM until such time that it runs out of cards or something else goes wrong. When this occurs program control will pass to the next statement. This sort of construction is particularly useful when it is not known in advance how many sets of numbers are to be read in.

When exception conditions are handled in this manner action labels may not be used.

CHAPTER 5

BLOCKS

5.1 Introduction

A block has the general form:

```
BEGIN
   D;  ⎫
   D;  ⎬ one or more declarations
   D;  ⎭
   S;  ⎫
   S;  ⎬ one or more statements
   S   ⎭
END
```

Observe that blocks and compound statements are not the same. A compound statement has no declarations in it whereas a block has at least one.

A block is a statement and may be governed by an IF clause, or a FOR clause etc. like any other statement. Moreover a block may itself form one of the statements of another block or a compound statement. Hence blocks may be 'nested' within blocks.

Notice that the simple ALGOL program, program 1 of section 1.1, is an example of a block. More complicated ALGOL programs are also blocks but generally contain more blocks nested within them.

5.2 Variable Scope

When a block is entered new identifiers are brought into existence as the declarations at the start of the block are encountered. These identifiers only exist within the block and cease to exist as soon as the block is exited whether this is through the END or via a GO TO statement branching out of the block.

Unlike a compound statement a block may only be entered via the BEGIN for only in this way do the identifiers declared within the block come into existence. If it were permissible to jump straight into the statements in a block then attempts would be made to manipulate variables which did not exist! To prevent attempts to enter a block at any point other than at the BEGIN, labels must be declared in the innermost block in which they are used to label a statement. This means that a label cannot be accessed in an outer block and therefore a GO TO statement in an outer block cannot make use of a

label in an inner block.

Example

```
        BEGIN
           INTEGER J;
           LABEL K;
           BEGIN
              INTEGER I;
              LABEL L;
           L: READ (CARD, /,I);
              IF I = 10 THEN GO TO K
           END;
           J := 3;
     K:  END;
```

Inner

Outer

The label L in the inner block must be declared in the inner block and cannot be declared alongside the label K in the outer block. The GO TO K statement in the inner block which causes program control to jump out of the inner block, is perfectly permissible because the inner block is part of the outer block in which the label K appears and as part of the outer block it may access any labels appearing in it.

A possible ambiguity arises when one identifier is declared twice in such a way that the scope of the first declaration overlaps with the scope of the second. This can happen when the second declaration occurs in a block nested within the block containing the first declaration. In such circumstances the declaration corresponding to any occurrence of the identifier is the most recent declaration which is still current.

Example

```
BEGIN
   INTEGER I;
   I := 3;
   BEGIN
      INTEGER I;
      I := 4;
      WRITE (LP,/,I)
   END;
   WRITE (LP,/,I)
END;
```

Inner

Outer

The integer I, declared in the outer block remains in existence throughout the whole of the outer block. The integer I, declared in the inner block exists only within the inner block and all occurrences of I within the inner block are taken to refer to that integer I, declared in the inner block since that is the most recent declaration of I. On exiting from the inner block the I declared within it ceases to exist but the I declared in the outer block is still valid and all subsequent occurrences of I are taken to refer to the I declared in the outer block.

Thus the output from the above example would be

```
4,
3,
```

because the first WRITE statement encountered is in the inner block

and the I being output is taken as being the I declared in the inner block which was assigned the value 4. After exiting from the inner block the second WRITE statement is encountered which outputs the value 3 because the I which held the value 4 no longer exists and the most recent, valid declaration of I is that at the top of the outer block. It is this I which is assigned a value of 3 and is written out in the second WRITE statement.

Variables that are declared within a block are said to be 'local' to that block. Variables that are accessible to an inner block but are declared in an outer block are said to be 'global' to the inner block.

.3 Dynamic Arrays

It is conceivable that an array local to a given block may be required to have different dimensions on each occasion that the block is utilised. For example in a program which processes the exam. results of a class of pupils, the array containing their marks need only be as large as the number of pupils in the class. Thus if the one program is being used to analyse all the exam. results of every class it would be desirable to change the array dimensions in accordance with each class size. ALGOL allows the programmer to do this by using variables in the bound pair list of the array declaration. There is one restriction in doing this, and that is that the values of the variables must be known at the point where the declaration is made.

Example 1

```
BEGIN
   INTEGER I, J;
   READ (CARD,/,I,J);
   BEGIN
      REAL ARRAY A [1 : I, 0 : J];
        .
        .
        .
   END
END;
```

The array declaration in this example is valid because the values of I and J appearing in the bound pair list will be known on encountering the declaration since they are read in, in the READ statement.

Example 2

```
BEGIN
   INTEGER I, J;
   REAL ARRAY A [1 : I, 0 : J];
   I := 5; J := 6;
   .
   .
   .
END;
```

This time the array declaration is invalid since the values of I and J are not known at the point of declaration.

Example 3

```
BEGIN
   INTEGER I, J;
   I := 5; J := 6;
   REAL ARRAY A [1 : I, 0 : J];
   .
   .
   .
END;
```

In this example the array declaration is invalid because it follows a statement. Declarations must precede all statements - see section 5.1.

Once an array has been declared its dimensions remain constant throughout the block in which it was declared and cannot be made to vary within that block by subsequently altering the value of the variables used in its bound pair list.

CHAPTER 6

PROCEDURES

Introduction

A procedure declaration in effect associates a portion of code with an identifier. At run time all occurrences of the procedure identifier will appear to be replaced by this code. For instance it may be necessary at several points in a program to add 2 arrays A and B together and output the result. Instead of writing out the piece of code for this on each occasion, the appropriate code can be associated with an identifier ADD, say, and it is then sufficient to write ADD wherever the arrays have to be added together and output.

Example The procedure ADD might be declared as follows:

```
PROCEDURE ADD;
BEGIN
   INTEGER I;
   FOR I := 1 STEP 1 UNTIL 10 DO A[I] := A[I] + B[I];
   WRITE(LP,  < 10I6 >, A[*])
END;
```

With this declaration, the following code in part (A) is exactly equivalent to the code in part (B).

(A)
```
FOR J := 1 STEP 1 UNTIL 10 DO A[J] := J;
READ (CARD, /, B[*]);
ADD;
READ (CARD, /, A[*]);
```

(B)
```
FOR J := 1 STEP 1 UNTIL 10 DO A[J] := J;
READ (CARD, /, B[*]);
BEGIN
   INTEGER I;
   FOR I := 1 STEP 1 UNTIL 10 DO A[I] := A[I] + B[I];
   WRITE(LP,  < 10I6 >, A[*])
END;
READ (CARD, /, A[*]);
```

The piece of code associated with the procedure identifier is called the 'procedure body'. In the procedure ADD above, the procedure body is

```
BEGIN
   INTEGER I;
   FOR I := 1 STEP 1 UNTIL 10 DO A[I] := A[I] + B[I];
   WRITE(LP, < 10I6 > , A[*])
END
```

i.e. everything between BEGIN and END inclusively.

In general the procedure body can be a simple statement, compound statement or a block. The compound statement and block may of course be nested. When the procedure body is a block the scope of all the variables declared within it follow the rules as detailed in section 5.2.

The 'procedure heading' is that part of the procedure which precedes the procedure body. In the above example the procedure heading is

```
PROCEDURE ADD;
```

Procedure headings are frequently more complicated than this as will be seen in the following sections.

A procedure declaration can therefore be summarised as a procedure heading followed by a procedure body.

A reference to a procedure in a program is known as a 'procedure call'. In part (A) of the above example, the third line (i.e. ADD;) is an example of a procedure call.

Note that a procedure declaration is just a declaration and no statement in it is executed until the procedure is called during program execution.

6.2 Procedures with Parameters

These provide a means whereby the code associated with the procedure identifier can be made to vary. A parameter list containing what are called 'formal parameters' is associated with the procedure identifier and the code is written so that it operates on these parameters. The formal parameters in the formal parameter list are separated by commas and enclosed in round brackets.

Example

```
PROCEDURE CHANGE(A, B);
REAL A, B;
BEGIN
   REAL S;
   S := A;
   A := B;
   B := S
END OF CHANGE;
```

The procedure identifier is CHANGE,
the formal parameter list is (A, B),
the formal parameters are A and B,
the procedure heading is lines 1 and 2,
the procedure body is lines 3 - 8,
"OF CHANGE" following the END is just a comment to indicate the end of the procedure.

A procedure with parameters is called in a program by the occurrence of the procedure identifier as before, but this must be followed by a list of actual parameters corresponding to the formal parameters in the procedure declaration. The formal parameters only mark the positions in the procedure body where the actual parameters have to be put when the procedure is called. There is a 1 - 1 correspondence between the formal and actual parameters and the actual parameters should be of the same type as the formal parameters. Line 2 of the procedure CHANGE above i.e. REAL A, B; specifies the type of the formal parameters and is called the 'specification part'. It indicates that when the procedure is called the actual parameters will be of type real. It is not a declaration - merely a specification.

Example If X, Y are variables having type real then CHANGE (X, Y) will cause the values of X and Y to be swapped over. CHANGE (X, Y) is equivalent to:

```
BEGIN
   REAL S;
   S := X;
   X := Y;
   Y := S
END;
```

That is, the code executed by the computer is the code of the procedure body but with the actual parameters, X and Y, replacing the formal parameters, A and B.

Summarising this : when a procedure with parameters is called, the actual parameters are used in the procedure body instead of the formal parameters.

6.3 Name and Value

In the last example the parameters are called by name, i.e. the names of the variables in the actual parameter list are used in the procedure body to replace the formal parameters. When parameters are called by name it is possible for the procedure to alter the value of the actual parameters as does in fact happen when CHANGE (X, Y) is executed, for example. Parameters can also be called by value however, and in this case only the values of the actual parameters are used in the procedure body not the variables themselves. Therefore when variables are called by value, the actual parameters always possess the same values after the procedure call as they had immediately before it.

Consider the procedure SUM where the parameters are called by name:

```
PROCEDURE SUM (A, B);
INTEGER A, B;
BEGIN
   A := A + B;
   WRITE(LP, /, A)
END;
```

If X and Y are integers such that X = 5 and Y = 6 then SUM (X, Y) is equivalent to

```
BEGIN
   X := X + Y;
   WRITE(LP, /, X)
END;
```

i.e. SUM (X, Y) will cause the value 11 to be output and will leave X holding the value 11.

Now consider the procedure where the parameters are called by value.

```
PROCEDURE SUM (A, B);
VALUE A, B;
INTEGER A, B;
BEGIN
   A := A + B;
   WRITE(LP, /, A)
END;
```

This time SUM (X, Y) will have the value 11 output as before but will leave X holding the value 5 that it started with. Thus call by value can be considered to be a sort of protection device for the actual parameters as their values can be used but not changed. The way in which this is accomplished is as follows.

When parameters are called by value, replacement variables of the same type as the formal parameters are declared by the compiler. These replacement variables are local to the procedure and are assigned the values of the actual parameters as soon as the procedure is called. It is these replacement variables that are manipulated by the procedure.

Example When the parameters are called by value the procedure SUM is effectively equivalent to the following piece of code:

```
PROCEDURE SUM (A, B);
INTEGER A, B;
BEGIN
   INTEGER A1, B1;
   A1 := A;
   B1 := B;
   A1 := A1 + B1;
   WRITE(LP, /, A1)
END;
```

The replacement variables in this case are denoted by A1 and B1. When the procedure is called, the actual parameters are used instead of the formal parameters A, B and the replacement variables A1, B1 are left alone.

Thus SUM (X, Y) is equivalent to

```
BEGIN
   INTEGER A1, B1;
   A1 := X;
   B1 := Y;
   A1 := A1 + B1;
   WRITE(LP, /, A1)
END;
```

Clearly this leaves the value of X and Y unchanged yet uses their values.

Therefore, unless a procedure is required to change the values of the actual parameters rather than just use them it is normal to call the parameters by value and not name. This is done in the 'value part' of a procedure heading and it precedes the specification part. For instance the value part of the following procedure heading is the second line.

```
PROCEDURE SUM(A, B);
VALUE A, B;
INTEGER A, B;
```

Notes

1. Every formal parameter must appear in the specification part; the value part specifies which of the formal parameters are to be called by value. It is possible for some of the parameters to be called by value and some by name.

2. Any variables used in the procedure which are neither declared nor specified in the procedure must be declared before the procedure is declared.

3. Arrays can only be called by name.

Example

```
PROCEDURE P2(A, B, C, D);
VALUE A, B;
REAL A, B, C;
REAL ARRAY D[*, *, *];
BEGIN
   .
   .
   .
END;
```

In the specification part, the bound-pairs of the array D may be replaced either by asterisks as in this case with one asterisk for each bound pair or by the lower bounds of the bound-pairs of the actual array to be used if these are known. In this case here the actual array used when the procedure is called must be 3-dimensional.

4. Arithmetic expressions may be used as the actual parameters of a procedure where the parameters are called by value. For example in SUM (A, B) with A, B called by value as above, SUM (5, 6) will output the value 11.

5. A failure will occur if a formal parameter called by name is used on the left side of an assignment and the corresponding actual parameter is not a variable. For example in SUM (A, B) with A, B called by name, SUM (5, 6) will cause a failure since in the procedure body, A appears on the left of an assignment and the corresponding actual parameter, 5, is not a variable. That is, a failure will result because the procedure will attempt to assign a value to 5 !

6. If a formal parameter never occurs on the left of an assignment statement in the procedure body then it may be called by name or value. If the actual parameter is an expression however, a call by name involves a fresh evaluation every time the parameter is mentioned while a call by value involves just one evaluation when the procedure is first called and is therefore usually more efficient.

Observe that if the parameters of the procedure CHANGE in the last section (6.2), are called by value then nothing is accomplished. This can be seen by considering the effect of calling CHANGE (X, Y) when the formal parameters were called by value. The effective code would be:

```
BEGIN
   INTEGER A1, B1;
   REAL S;
   A1 := X;
   B1 := Y;
   S := A1;
   A1 := B1;
   B1 := S
END
```

This leaves X and Y unchanged.

6.4 Type Procedures

Type procedures - or function procedures - are procedures, with or without parameters, which return a value when called and can therefore be used as ordinary variables. A few built-in type procedures have already been encountered as arithmetic intrinsics. For example SIN(X) and COS(X) are calls on the type procedures SIN and COS respectively both of which have one real parameter. In the case of SIN(X) the value returned is the sine of X.

The procedure heading of a type procedure has the general form

```
type PROCEDURE identifier (formal parameter list);
value part;
specification part;
```

where the 'formal parameter list' is optional; the 'identifier' can be chosen by the programmer; and the 'type' can be REAL, INTEGER or BOOLEAN.

The value returned by a type procedure is assigned to the procedure identifier in the procedure body but no assignment may be made to the procedure identifier outside the procedure body.

Example

```
INTEGER PROCEDURE FACTORIAL (N);
VALUE N;
INTEGER N;
BEGIN
   INTEGER I, PRODUCT;
   PRODUCT := 1;
   FOR I := 2 STEP 1 UNTIL N DO PRODUCT := PRODUCT * I;
   FACTORIAL := PRODUCT
END OF FACTORIAL;
```

In this example the value assigned to the procedure identifier FACTORIAL is factorial N.

Since a type procedure returns a value the procedure may be called in most places in which an ordinary variable can occur.

Examples

1. `A := FACTORIAL (5);` which is equivalent to `A := 120;`
2. `WRITE (LP, /, FACTORIAL (3));` which is equivalent to `WRITE (LP, /, 6);`

Observe that
```
A := FACTORIAL (3);
B := FACTORIAL (A);
```
is the same as `B := FACTORIAL (FACTORIAL (3));`

Thus the procedure itself may be used as its argument. It is good practice to ensure that the formal and actual parameters are of the same type although any necessary conversions from real to integer will be performed automatically. In the case of FACTORIAL the formal parameter and the result are both of type integer and thus FACTORIAL (3) may safely be used as the actual parameter of FACTORIAL.

5 Forward Procedure Declarations

Before a procedure may be referred to in a program it must be declared. Consider the case, however, where there are two procedures each of which contains a reference to the other in its procedure body. No matter which procedure is declared first it will contain a reference to an undeclared procedure. To enable the ALGOL compiler to handle this situation, provision has been made for 'forward procedure declarations'.

Example 1

```
PROCEDURE ONE; FORWARD;
```

Example 2

```
REAL PROCEDURE P (A, B, C);
VALUE A, B; REAL A, B, C;
FORWARD;
```

These simply tell the compiler that procedures ONE and P will be declared fully later on and in the meantime they may be used in the body of another procedure. If a full declaration is not made later on, an error will result.

CHAPTER 7

FURTHER FEATURES OF B6700 ALGOL

7.1 Multiple Assignments

Recall that an arithmetic assignment has the general form:

```
variable := arithmetic expression ;
```

and a Boolean assignment has the general form:

```
variable := Boolean expression ;
```

Now the arithmetic and Boolean assignments are considered to be arithmetic and Boolean expressions respectively, also, and consequently several assignments can be made in one statement.

Example 1

```
A := B := 3 ;
```

B := 3 is just a simple assignment. A := B := 3 has the same effect on A as A := 3 since B := 3 is considered to be an arithmetic expression with value 3.

Example 2 Similarly

```
OK := FLAG := TRUE ;
```

assigns the value TRUE to OK and to FLAG.

Example 3

```
A := 5 + B := 3 ;
```

Again B := 3 is just a simple assignment and the value of B := 3 is 3. Therefore the value of 5 + B := 3 is 5 + 3 = 8 and hence A is assigned the value 8.

Example 4

```
A := B := C := 3 + D := N ;
```

Here D is assigned the value of N and A, B, C are assigned the value N + 3.

The value of assignment statements may also be used as the subscript of a subscripted variable.

Example 5

```
A[I := 3] := 4 ;    is equivalent to

I := 3 ;
A[3] := 4 ;
```

Note that the assignment in the subscript is made before the assignment to the subscripted variable.

Example 6

```
I := 0 ;
THRU 10 DO WRITE (LP, /, B[I := I + 1, *]) ;
```

This will write out B[1, *], B[2, *], ..., B[10, *].

Example 7

```
Y := 1 ;
X := IF OK THEN 2 ELSE Y := Y + 1 ;
```

This is equivalent to

```
Y := 1 ;
IF OK THEN X := 2 ELSE X := Y := Y + 1 ;
```

7.2 CASE Statements

The general form of a CASE statement is

CASE AE OF case body ;

The 'case body' is a compound statement which may include any combination of simple statements, compound statements and blocks. 'AE' is an arithmetic expression which is integerised by rounding if not already integral.

The N statements within the case body are numbered from 0 to N-1 and the statement indicated by the arithmetic expression is executed. Only one statement from the case body is executed but this may be a compound statement or a block. After a statement in the case body has been executed program control passes to the statement following the CASE statement.

Example

```
CASE X OF
BEGIN
   I := 1 ;          0
   I := 2 ;          1
   BEGIN
      I := 33 ;      2
      J := 5
   END ;
   J := 16           3
END ;
K := 5 ;
```

The case body in this example consists of 4 statements of which one is a compound statement. If X = 0 then statement 0 is executed; if X = 1 then statement 1 is executed; if X = 2 then statement 2 is executed i.e. assignments are made to I and J; if X = 3 then statement 3 is executed. After whichever statement is executed the next statement to be executed will be K := 5.

If the value of the AE lies outside the range 0 to N - 1 then an error will occur.

7.3 CASE Expressions

The general form of a CASE expression is

```
CASE AE OF (list of expressions)
```

The N expressions in the 'list of expressions' are separated by commas and are numbered from 0 to N - 1. 'AE' represents an arithmetic expression and has the same rules as the AE in CASE statements. When a CASE expression is utilised only the one appropriate expression from the list is evaluated.

Example

```
A := CASE X OF (5, SQRT(5), SQRT (SQRT(5)));
```

If X = 0 then A is assigned the value 5;
If X = 1 then A is assigned the square root of 5;
If X = 2 then A is assigned the square root of the square root of 5.

Note that a case expression may not be used on the left of an assignment operator, e.g. it is incorrect to write

```
CASE X OF (A, B, C) := 6;
```

7.4 FILL statements

The general form of a FILL statement is

```
FILL array row WITH value list;
```

The 'value list' contains the list of values (separated by commas) which are to be used to fill the array row. The value list may contain a repeat part, e.g. 7 (0, 1, 2), where the part in brackets is repeated the number of times indicated by the unsigned integer outside the brackets.

Examples

```
FILL A[*] WITH 4(1, 3, 5, 7, 9), -5;
FILL B[2, *] WITH 28(3);
```

If the value list contains more values than the array row, the extra values are ignored. If the list contains too few values then filling stops at the end of the list.

7.5 The Dummy Statement

The dummy statement instructs the computer to do absolutely nothing.

Example

```
IF X>0 THEN ELSE X := 2;
```

The dummy statement exists here between THEN and ELSE. The effect of the whole statement is: if X>0 then do absolutely nothing, otherwise assign X the value 2.

7.6 Switches

Switches allow the programmer to dynamically select one of several alternatives at any given point in program execution. There are 4 sorts of switches:

1. Lists
2. Formats
3. Files
4. Labels

7.6.1 List, Format and File Switches

The declaration of these switches have the general form:

```
SWITCH { LIST   } identifier := list of elements;
       { FORMAT }
       { FILE   }
```

where only one of LIST, FORMAT and FILE is chosen. The 'identifier' is chosen by the programmer like any other identifier. The 'list of elements' is a list of lists, formats or files depending on the type of switch. The beginner is unlikely to require SWITCH FILES which are therefore omitted from the present discussion.

Examples

1.
```
LIST LO(A, B, C), L1(D, E);
SWITCH LIST SL := LO, L1;
```

2.
```
FORMAT F1(X5, I5);
SWITCH FORMAT SF := (X6, I6), (X10, F3.1), F1;
```

The lists in SWITCH LIST declarations must all have been previously declared as in example 1. If a format identifier is used in a SWITCH FORMAT declaration then it must have been previously declared. For instance, in example 2, F1 is a format identifier and must therefore be declared before it can be used in the SWITCH FORMAT declaration. Formats can be included directly however, by enclosing the required format in round brackets.

The N elements in the list of elements in the declaration are numbered from 0 to N-1 going from left to right. Thus using the 2 declarations above,

```
SL[0] is LO  or A, B, C
SF[2] is F1  or X5, I5
```

Example

If a SWITCH LIST, SL, is declared which contains 10 lists then they may all be read in by the statement

```
FOR I := 0 STEP 1 UNTIL 9 DO
READ (CARD, /, SL[I]);
```

Similarly if a SWITCH FORMAT, SF, is declared which contains 10 formats - one for each list - then it too may be used in the READ statement:

```
FOR I := 0 STEP 1 UNTIL 9 DO
READ (CARD, SF[I], SL[I]);
```

If the subscript of a switch variable is not integral then it is integerised by rounding. In the case of list and format switches, if the subscript lies outside the range 0 to N-1 then an error will occur.

7.6.2 Label Switches

The general form of these switches is

```
SWITCH identifier := list of labels;
```

Observe that only the word SWITCH appears before the 'identifier'. The 'list of labels' consists of labels separated by commas and all the labels must have been previously declared.

Example

```
LABEL L1, L2, L3;
SWITCH S := L1, L2, L3;
```

The N labels in the list of labels are numbered from 1 to N going from left to right. (Not 0 to N-1 as with the other switches). Thus

```
S[1] is L1
S[2] is L2
S[3] is L3.
```

If the subscript is not integral then it is integerised by rounding. If the subscript lies outside the range 1 to N an error does not occur as might be expected; instead the instruction attempting to branch to the invalid label is ignored and control passes to the next instruction.

Example

```
LABEL L1, L2, L3;
SWITCH S := L1, L2, L3;
     :
     :
GO TO S[4];
K := K + 1;
     :
     :
```

When the statement 'GO TO S[4]' is encountered it is ignored since S[4] is an invalid label and control passes to the next statement i.e. K := K + 1;

A switch label list may include conditional elements.

Example

```
LABEL L1, L2, L3;
SWITCH S := IF FLAG THEN L1 ELSE L2, L3;
```

S[2] is L3;
S[1] depends on the value of FLAG. If FLAG is TRUE then S[1] is L1 and if FLAG is FALSE then S[1] is L2.

The elements of a switch label list may also be controlled by a CASE expression.

Example

```
LABEL L1, L2, L3, L4;
SWITCH S := CASE N OF (L1, L2, L3), L4;
```

S[2] is L4,
S[1] depends on the value of N and may take the values L1, L2 or L3.

7.6.3 Forward Switch Declarations

The list of elements of each of the 4 sorts of switches may include elements of other switches.

Example

```
FORMAT F1(X5, I6);
SWITCH FORMAT SF := F1, (I3, X2, F3.1), F1;
SWITCH FORMAT SF2 := SF[1], (X5, I7);
```

SF[1] is a legitimate element of the SWITCH FORMAT SF2.

Since switch elements may include elements of other switches it is possible to have two switches each of which includes an element of the other. No matter which switch is declared first it will contain a reference to an undeclared switch element. Hence the need for forward declarations.

The general form of a forward switch declaration is

```
SWITCH type identifier FORWARD;
```

where 'type' can be LIST, FORMAT or FILE. If type is omitted then a label switch is assumed.

These forward declarations tell the compiler that a switch declaration will be made in full later on and when the second switch declaration makes use of the first switch, the compiler recognises it even though it has not been fully declared.

7.7 OWN Declarations

Normally the values of variables declared in a block are lost on exiting from the block. The OWN declaration causes the value of a variable to be retained so that on a subsequent entry to the block in which it is declared, the variable has the same value as it had at the last exit from the block. A variable can be declared to be an OWN variable by prefixing its usual declaration with the word OWN.

Example

```
OWN INTEGER F;
OWN REAL ARRAY SUM[1 : 5];
```

Parameters of a procedure may not be specified as being OWN variables.

As with all other variables, OWN variables may only be accessed within the block in which they are declared although with regard to their values they behave as if they had been declared in the program's outermost block.

Since an OWN identifier is local to the block in which it occurs no initial value can be assigned to an OWN variable before entry to that block. This gives rise to a problem in initialising an OWN variable. If an assignment is made at the start of the block e.g. F := 0; then this will be executed at every entry to the block and so prevent the attempt to save the value of F. One solution of this problem is to enclose the initialisation of the variables in a compound statement controlled by a Boolean variable in such a way that the compound statement is only executed when the Boolean variable has the value TRUE. Assign the Boolean variable the value TRUE before the block is entered for the first time and FALSE thereafter so that initialisation will only be carried out once.

7.8 DEFINES

A DEFINE declaration associates some ALGOL source language text with a DEFINE identifier. At compile time (i.e. before the program is executed) every occurrence of each DEFINE identifier is replaced by its associated ALGOL source language text. Note that this is not the same as a procedure which cannot be invoked until execution time. Furthermore the text associated with a DEFINE identifier need not be a statement whereas a procedure body must be a statement. Like a procedure, a DEFINE may or may not have parameters.

The general form of a DEFINE declaration without parameters is

```
DEFINE identifier = text #;
```

where the 'identfier' can be chosen by the programmer and the 'text' is the ALGOL source language text which becomes associated with the DEFINE identifier. The # symbol is used to terminate the text but is not part of it. As with other declarations several DEFINE identifiers may be declared at once by separating them with commas after the #.

Example

Suppose the following 3 DEFINE declarations are made:

```
DEFINE FIRSTLIST = A, B , C, D, E, F  #,
       SECONDLIST = P, Q, R, S, T, U, V, W  #,
       INITIALISE = A := 10;
                    B := 3;
                    C := 16 #;
```

With these declarations the following piece of code

```
INITIALISE;
WRITE (LP, /, FIRSTLIST, SECONDLIST);
READ (CARD, /, SECONDLIST);
```

is equivalent to:

```
A := 10;
B := 3;
C := 16;
WRITE (LP, /, A, B, C, D, E, F, P, Q, R, S, T, U, V, W);
READ (CARD, /, Q, R, S, T, U, V, W);
```

The general form of a DEFINE declaration with parameters is

```
DEFINE identifier (list of formal symbols) = text #;
```

where the 'list of formal symbols' is a list of identifiers separated by commas. If a DEFINE identifier has been declared as a DEFINE with parameters then each occurrence of the DEFINE identifier must be followed by a list of 'actual texts' separated by commas and enclosed in round brackets. At compile time the text associated with the DEFINE identifier replaces each occurrence of the DEFINE identifier but with the actual texts replacing the formal symbols except in COMMENTS.

Notes

1. A DEFINE identifier may not appear in a FORMAT declaration nor in the editing specifications of a READ or WRITE statement.

2. The actual texts used with a DEFINE with parameters may not include unmatched bracketing i.e. a left bracket without a right and vice versa. A comma may appear in the actual texts only between matching bracketing symbols.

Example

```
DEFINE OVER(I) = FOR I := 1 STEP 1 UNTIL N DO #;
```

Wherever the DEFINE identifier OVER appears the actual text enclosed in brackets will replace the formal symbol I in the above declaration. Thus

```
OVER(I) OVER(J) SUM := SUM + A[I, J];
```

is equivalent to

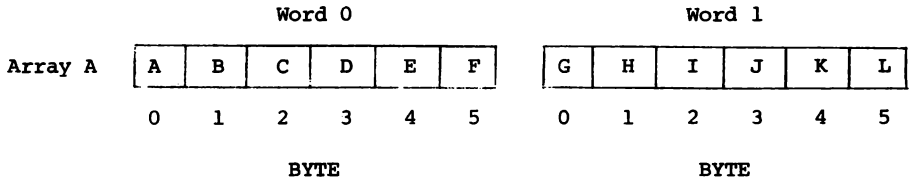
```
FOR I := 1 STEP 1 UNTIL N DO
FOR J := 1 STEP 1 UNTIL N DO
SUM := SUM + A[I, J];
```

CHAPTER 8

STRING HANDLING

8.1 Introduction

Character strings may be stored in arrays six characters (bytes) to an element (word). So that the character string "ABCDEFGHIJKL" would be stored in an array A[0 : 1] as:-



8.2 Arrays and Pointers

An array to be used for strings may be declared as

```
ALPHA ARRAY array name [lower bound : upper bound];
```

However, the machine treats the array as if it were an array of type real and thus a declaration

```
ALPHA ARRAY A[0 : 1]; or
      ARRAY A[0 : 1];
```

would suffice.

In order to fix some position in the string we need to point to the byte of the word of the array which corresponds to the position and hence we need to declare a variable declared as a pointer which will be assigned the relative address of the character position with respect to the beginning of the array row. We must thus have a declaration

```
POINTER  list of identifiers;
```

Example

```
POINTER PA;
```

8.3 Pointer Assignments

Having declared a pointer we must now assign it some "value", this may be achieved by using an expression

```
POINTER (array name [subscript])
```

Example

```
PA := POINTER (A[0]);
```

This "points" PA at the first character in element zero of the array A, that is byte zero word zero.

We may achieve this more simply by writing

```
PA := POINTER (A);
```

which assumes the zero element and "points" PA to the same position as in the above example.

If we wish to "point" the pointer to some other character in the array then we may write, for example:

```
PA := POINTER (A) + 3;

ARRAY A:   A B C D E F    G H I J K L
                 ↑
                PA

PA := POINTER (A[1]) + 4;

ARRAY A:   A B C D E F    G H I J K L
                                 ↑
                                PA

PA := POINTER (A) + 10;

ARRAY A:   A B C D E F    G H I J K L
                                 ↑
                                PA
```

In general we have

```
pointer := POINTER (array name [subscript])

                ± (arithmetic expression);
```

Note that, except for single identifiers or constants, the arithmetic expression must be enclosed in parentheses. The arithmetic expression is rounded to an integer if it does not have an integer value.

For the remainder of this chapter we shall assume the following:-

Definitions

SB source string before execution
SA source string after execution
DB destination string before execution
DA destination string after execution
Ø the character "blank"

Declarations

```
ARRAY A,B[0 : 13];
POINTER PA, PB, PC, PD;
```

We shall assume at the start of each example that A contains the string

```
ABCDEF    GHIJKL    MNOPQR    STUVWX.....
```

and B contains blanks, unless otherwise stated.

The following further illustrates the use of pointers

```
   Y := 3;
   X := 2;
  PA := POINTER (A) + (X * Y);
  PC := PA + 4;
  PD := IF X<4 THEN PA + 2 ELSE PA + Y;
```

```
Array A:  A B C D E F   G H I J K L   M N O P Q R   S T U V W X.....
                        ↑   ↑   ↑
                       PA   PD  PC
```

8.4 Updating Pointers

As character strings are operated upon it is often necessary to keep track of where one is in the string and when using REPLACE and SCAN statements (see below) we may use the following updating technique:

```
PA : PC
```

This is embedded in a REPLACE or SCAN statement and sets PA initially equal to PC. As the statement is performed PA is updated to the current character position; PC remains unchanged. We shall see how to use this more clearly when we meet REPLACE and SCAN statements.

8.5 REPLACE statement

REPLACE destination pointer BY source pointer FOR arithmetic expression units.

The units should be "CHARACTERS" or "WORDS" but if no units are specified then "CHARACTERS" will be assumed.

Examples

1.
```
PA := POINTER (A);
PB := POINTER (B);
REPLACE  PB  BY  PA  FOR  2  WORDS;
```

```
SB      A B C D E F    G H I J K L    M N O P Q R.....
        ↑
        PA
```

```
DB      ⌀ ⌀ ⌀ ⌀ ⌀ ⌀    ⌀ ⌀ ⌀ ⌀ ⌀ ⌀    ⌀ ⌀ ⌀ ⌀ ⌀ ⌀.....
        ↑
        PB
```

```
SA      A B C D E F    G H I J K L    M N O P Q R.....
        ↑
        PA
```

```
DA      A B C D E F    G H I J K L    ø ø ø ø ø ø.....
        ↑
        PB
```

2. `REPLACE PD:PB BY PC:PA FOR 8;`

(Note "CHARACTERS" will be assumed since units are omitted.)

```
SB      A B C D E F    G H I J K L    M N O P Q R.....
        ↑
        PA

DB      ø ø ø ø ø ø    ø ø ø ø ø ø    ø ø ø ø ø ø.....
        ↑
        PB

SA      A B C D E F    G H I J K L    M N O P Q R.....
        ↑                  ↑
        PA                 PC

DA      A B C D E F    G H ø ø ø ø    ø ø ø ø ø ø.....
        ↑                  ↑
        PB                 PD
```

3. `REPLACE PA + 4 BY PB FOR 10;`

```
SB      ø ø ø ø ø ø    ø ø ø ø ø ø    ø ø ø ø ø ø.....
        ↑
        PB

DB      A B C D E F    G H I J K L    M N O P Q R.....
        ↑
        PA

SA      ø ø ø ø ø ø    ø ø ø ø ø ø    ø ø ø ø ø ø.....
        ↑
        PB

DA      A B C D ø ø    ø ø ø ø ø ø    ø ø O P Q R.....
        ↑
        PA
```

4. `REPLACE PC:PB+4 BY PA:PA+2 FOR 3;`

```
SB      A B C D E F    G H I J K L    M N O P Q R.....
        ↑
        PA

DB      ø ø ø ø ø ø    ø ø ø ø ø ø    ø ø ø ø ø ø.....
        ↑
        PB

SA      A B C D E F    G H I J K L    M N O P Q R.....
                ↑
                PA

DA      ø ø ø ø C D    E ø ø ø ø ø    ø ø ø ø ø ø.....
        ↑                  ↑
        PB                 PC
```

8.6 Truthsets

It is possible to set up an array which contains information as to whether a particular character belongs to a truthset or not. Truthsets are declared and characters assigned to them at the same time.

```
TRUTHSET THERE ("AEIOU*;,.");
```

This produces an array called THERE which contains the information that the characters "A E I O U * ; , ." are included in it; all other characters are excluded.

The B6700 has some "built-in" truthsets. The only one initially to be concerned about is ALPHA, this contains the characters

"A thru Z", "0 thru 9".

8.7 Conditions

WHILE and UNTIL

These function in a similar way to the WHILE and UNTIL clauses already met except that the left hand part of the Boolean expression is implied.

Examples

1.
```
REPLACE  PB  BY  PA  UNTIL  EQL  "E";
```

This replaces the string in B by the string in A until an "E" is reached.

2.
```
REPLACE  PB  BY  PA  WHILE  NEQ  "J";
```

This replaces the string in B by the string in A until a "J" is reached.

3.
```
SB   A B C D E F    G 1 2 3 4 5    6 7 8 9 J K.........
     ↑
     PA

     REPLACE  PB  BY  PA  WHILE  LSS  "4";

DA   A B C D E F    G 1 2 3 ⌀ ⌀    ⌀ ⌀ ⌀ ⌀ ⌀ ⌀ ⌀.........
     ↑
     PB
```

(The collating order in ascending order is "A thru Z", "0 thru 9")

WHILE IN and UNTIL IN

```
WHILE IN truthset

UNTIL IN truthset
```

We can, using these conditions, test for the occurrence of a set of characters in a string; the first occurrence of any of the characters returns TRUE.

Example

```
SB      A B C D E F   G H ; I J K   L * M N $ P
        ↑
        PA

        TRUTHSET  TAB ("K*$"), TAB1 ("*$");

1.      REPLACE  PB  BY  PA  UNTIL  IN  TAB;

DA      A B C D E F   G H ; I J ∅   ∅ ∅ ∅ ∅ ∅ ∅.....
        ↑
        PB

2.      REPLACE  PB  BY  PA  WHILE  IN  ALPHA;

DA      A B C D E F   G H ∅ ∅ ∅ ∅   ∅ ∅ ∅ ∅ ∅ ∅.....
        ↑
        PB

3.      REPLACE  PB  BY  PA  UNTIL  IN  TAB1;

DA      A B C D E F   G H ; I J K   L ∅ ∅ ∅ ∅ ∅.....
        ↑
        PB
```

8.8 SCAN Statement

This statement "scans" a string until a condition is satisfied.

Example

```
        SCAN  PC:PA  UNTIL  GTR  "K";

SB      A B C D E F   G H I J K L   M N O P Q R.....
        ↑
        PA

SA      A B C D E F   G H I J K L   M N O P Q R.....
        ↑                     ↑
        PA                    PC
```

Exactly the same conditions used with REPLACE (WHILE, UNTIL, WHILE IN, UNTIL IN) may be used in SCAN statements.

8.9 Maximum Count and Toggle

In a REPLACE or a SCAN statement we may use a count of the number of characters to be transferred to terminate the REPLACE or SCAN. This is achieved with a count written as

FOR arithmetic expression units.

"Units" may be omitted, in which case CHARACTERS is assumed.

Example

```
1.      REPLACE  PB  BY  PA  FOR 3;

DA      A B C ∅ ∅ ∅    ∅ ∅ ∅ ∅ ∅ ∅.........

2.      SCAN  PC:PA  FOR  6  UNTIL  EQL  K;

SB      A B C D E F    G H I J K L    M N O P Q R.........
        ↑
        PA

SA      A B C D E F    G H I J K L    M N O P Q R.........
        ↑              ↑
        PA             PC
```

The scan terminated after 6 characters since this was the "maximum count" specified.

There exists within the B6700 a "built-in" variable called TOGGLE which is "set" i.e. returns TRUE when the SCAN or REPLACE ceases on a maximum count and is "reset" i.e. returns FALSE otherwise. Every time a REPLACE or SCAN statement is used which does not terminate on a maximum count then TOGGLE is reset (FALSE). This enables the REPLACE and SCAN statements to be terminated either by a condition (WHILE, UNTIL, WHILE IN, UNTIL IN) or by a maximum count and allows the programmer to test immediately after the statement whether the condition or maximum count terminated the statement.

Example

```
        SCAN  PC:PA  FOR  10  WHILE  LSS  "H";
        IF  TOGGLE  THEN  ELSE
        REPLACE  PC:PC  BY  PB  FOR  1;

SB      A B C D E F    G H I J K L    M N O P Q R.........
        ↑
        PA

SA      A B C D E F    G H I J K L    M N O P Q R.........
        ↑                ↑
        PA               PC
```

After the scan TOGGLE is FALSE.

```
DB      A B C D E F    G H I J K L    M N O P Q R.........
        ↑                ↑
        PA               PC

SB      ∅ ∅ ∅ ∅ ∅ ∅    ∅ ∅ ∅ ∅ ∅ ∅
        ↑
        PB

DA      A B C D E F    G ∅ I J K L    M N O P Q R.........
        ↑                  ↑
        PA                 PC
```

8.10 Residual Count

It is possible to count how many characters have been scanned or replaced by using the following construction

FOR residual count maximum count

Residual count then contains the value of maximum count less the number of characters scanned or replaced.

Example (J is previously declared INTEGER)

```
1.      SCAN  PC:PA  FOR  J:10  UNTIL  EQL  "C";

SB      A B C D E F    G H I J K L    M N O P Q R.........
        ↑
        PA

SA      A B C D E F    G H I J K L    M N O P Q R.........
        ↑   ↑
        PA  PC
```

J now contains 10 - 2 = 8.

Thus residual count may be used to ensure that a REPLACE or SCAN statement does not move a pointer outside an array bounds. If an attempt is made to place a pointer outside an array, an error "SEG ARRAY ERROR" will occur and the program will terminate prematurely.

8.11 Compare Statements

Pointers may be used in Boolean expressions as follows:

PA = "B" returns TRUE if PA points to a "B",

returns FALSE if PA does not point to a "B".

Similarly for LSS, GTR, GEQ, etc. when the pointer is compared with a single character.

PA = "BAC" returns TRUE if PA points to the "B" in a string "BAC",

returns FALSE if PA does not point to the "B" in a string "BAC".

PA = PB FOR arithmetic expression units

returns TRUE if PA and PB point to an identical string for arithmetic expression units,

returns FALSE if PA and PB do not point to an identical string for arithmetic expression units.

"Units" may be omitted in which case CHARACTERS will be assumed.

Example

```
          IF  PA  =  PB  FOR  3  THEN  PA  :=  PB + 3;

SB        A B C D E F    G H I J K L    A B C E F G.........
          ↑                             ↑
          PA                            PB

SA        A B C D E F    G H I J K L    A B C E F G.........
                                        ↑     ↑
                                        PB    PA
```

Since PA and PB point at the same sequence of characters for three characters the statement following the THEN is performed.

PA IN truthset

returns TRUE if PA points at a character which is in the truthset,

returns FALSE if PA points at a character which is not in the truthset.

Example

```
          I := 0;
          IF  PA  IN  ALPHA  THEN
          BEGIN
             I := I + 1;
             PA := PA + 1
          END
          ELSE PA := PA + 2;

SB        A B C D E F    G H I J K L.........
          ↑
          PA

SA        A B C D E F    G H I J K L.........
            ↑
            PA
```

I now contains 1.

We may thus use pointer comparisons in IF, WHILE and DO-UNTIL statements.

8.12 DELTA Intrinsic

DELTA (pointer expression, pointer expression) returns the number of characters between the two pointer expressions, which may simply be pointers, as a positive integer if the first pointer expression is less than the second and as a negative integer if the first pointer expression is greater than the second.

8.13 INTEGER Intrinsic

INTEGER (pointer expression, arithmetic expression) returns the integer value of the characters starting at the pointer expression for the number of characters specified by the arithmetic expression.

Example

```
A:.      A B C D E 1    2 3 4 F G H.........
                 ↑
                 PA
```

INTEGER (PA,3) returns the integer 123.

8.14 Conversion of an Integer Variable to a Character String

The use of a units expression "DIGITS" in a REPLACE statement converts an integer to a character string and performs a replace.

Example (I is declared INTEGER)

```
          I := 63;
          REPLACE  PA  +  8  BY  I  FOR  3  DIGITS;

SB        A B C D E F    G H I J K L    M N O P Q R.........
          ↑
          PA

SA        A B C D E F    G H Ø 6 3 L    M N O P Q R.........
          ↑
          PA
```

8.15 Use of Pointers in I/O

We may read from and write to files using a pointer and an "A" format, the pointer must "point" to the location in the array we wish to read to or write from and the number of characters transferred is controlled by the format.

Example

```
BEGIN
   FILE CARD (KIND = READER),
   LP (KIND = PRINTER);
   ARRAY BUFF[0 : 13];
   POINTER PB;
   PB := POINTER (BUFF);
   READ (CARD, <A80>, PB);
   WRITE (LP, <A80>, PB)
END.
```

This program would read 80 characters from the card reader and write 80 characters to the line printer.

Omissions

For the sake of simplicity many features of B6700 ALGOL have been completely omitted and many others have been deliberately simplified. For example, there are many more statements, intrinsic functions, and editing phrases than have been given here and important features such as double precision variables, file handling and bit manipulation have been entirely omitted. Details about these topics and many others can be found in the following documents.

1. BURROUGHS B6700/B7700 EXTENDED ALGOL LANGUAGE INFORMATION MANUAL.

2. USING POINTERS IN ALGOL ON B6700/B5700 COMPUTING SYSTEMS.

3. BURROUGHS B6700 SYSTEM MISCELLANEA.

APPENDIX

DEBUGGING

There are 3 sorts of errors which can occur in connection with a computer program.

1) Syntax errors,

2) Run time errors,

3) Logic errors.

Syntax Errors

These are mistakes in the program in which an illegal construct has been used, e.g. a missing semicolon or an undeclared identifier. The compiler automatically checks every program for syntax and writes out an error message whenever a syntax error is found. Unfortunately syntax errors can so confuse the compiler that any of the following actions is liable to occur:

1) The error is message is quite misleading,

2) One error is flagged with 2 or more quite different error messages together,

3) An error is flagged several lines after the actual error occurred (the error message may well be misleading also),

4) As a consequence of an earlier error subsequent "errors" are flagged where no errors actually occur,

5) As a consequence of an earlier error some subsequent errors escape unnoticed.

Thus when an error is flagged the programmer should

1) Check the immediate vicinity for the indicated error. If it does not occur then

2) Check the immediate vicinity for any error at all. If none is found then

3) Decide if the error is the result of a previous error. If it is then ignore the latest error message but if it is not then

4) Check through the previous lines of the program looking for any unflagged error.

To save time various checks should be performed on every program before it is punched. Check that

1) Every BEGIN has an END,

2) The last END is followed by a full stop,

3) Every identifier is declared,

4) Every declaration has a semicolon after it,

5) There is a semicolon after every statement which is followed by a statement,

6) No ELSE has a semicolon immediately before it,

7) No DO-UNTIL statement has a semicolon immediately before the UNTIL,

8) No reserved words are used as identifiers,

9) No identifier is declared twice at the head of any block,

10) If any of the restricted words given in section 1.2 are used as identifiers they are not used in their other capacities as well, e.g. SPACE, SKIP, SIN etc.,

11) Every "(" has a ")",

12) Every "[" has a "]",

13) For every open quotation marks there is a close quotation marks,

14) The symbol := is used in assignment statements and the symbol = in Boolean expressions,

15) Labels are declared in the innermost block in which they appear.

In addition it may be worthwhile to check that no identifier is declared but not used. Although this will not cause an error in itself it is likely that something has been left out by mistake. To ensure that no identifier is used which has not been declared it is a good idea to add to the declarations as identifiers are required during program writing. But do not forget to place all declarations at the head of their block.

After the program has been punched every card must be checked for punching errors and corrected as necessary.

Run Time Errors

These are mistakes in the program which cause the program to fail during execution rather than during compilation. Two frequent causes of run time errors are 1) trying to divide by zero and 2) trying to access an element of an array outside the declared bounds. When a run time error occurs, a short (but usually accurate) error message is given together with an indication of whereabouts in the program it took place.

To aid the programmer find the lines containing any errors there are 2 useful "options" which may be set at the start of a program. The option SEQ causes each line of the program to be numbered - this number being printed beside each line in the program listing - and the option LINEINFO which causes the line numbers of the last few lines executed to be given along with any run time error message. To set these options a card containing

```
$ SET SEQ LINEINFO
```

should be placed before the first BEGIN of the program. This is not an ALGOL statement and must not have a semicolon at the end.

To avoid as many run time errors as possible several checks should be made during program writing. Check that

1) Where a division is performed the denominator will never be zero,

2) Where the intrinsic SQRT is used its argument will never be negative,

3) Where the intrinsics LOG and LN are used their arguments will always be positive,

4) The subscripts in every subscripted variable will always be within the correct range for that variable,

5) All pointers are initialised before being used,

6) No pointer can possibly point beyond the end of an array.

There are many more causes of run time errors than those mentioned here but these 6 occur so frequently they deserve to be individually checked in every program. If it is not clear why a run time error is occurring it may be useful to output the values of the key variables immediately before the error occurs by inserting a WRITE statement.

Logic Errors

These are mistakes in the logic of a program. When a program compiles correctly and does not fail during execution due to run time errors but produces incorrect results the cause is usually faulty logic. Logic errors can be very hard to debug and unfortunately no error messages or line numbers are available to guide the programmer to the source of the trouble. Therefore when a program is not behaving as intended it should first of all be carefully examined line by line for possible faults. If none can be found it is a good idea to try to work through the program by hand using some test data following each instruction to the letter just as the computer would. If this is impractical or still fails to trace the fault WRITE statements can be inserted in various parts of the program which will indicate what route through the program is being taken and what values the key variables have at various points.

There are two useful options which may be set at compile time and which can be useful in debugging and improving a program. If the option XREF is set then a cross reference listing of all the identifiers in the program is provided; each identifier is listed together with the number of every line on which that identifier occurs. If the option STATISTICS is set then a listing is obtained which shows how many times each block (including procedures) was entered and how long was spent in each. The options XREF and STATISTICS may be set at the same time as the options SEQ and LINEINFO.

Example

```
$ SET SEQ LINEINFO XREF STATISTICS
```

INDEX